

Васильев А. Н.

САМОУЧИТЕЛЬ

C++

с примерами и задачами

4-е издание

(удовлетворяет

C++ 11 и C++ 14)



Наука и Техника
Санкт-Петербург
2016

Васильев А. Н.

Самоучитель С++ с примерами и задачами. 4-е издание (переработанное). Книга + виртуальный CD. — СПб.: Наука и Техника, 2016. — 480 с.: ил. (+ виртуальный CD)

Под редакцией Финкова М.В.

Серия «Самоучитель»

Данная книга является четвертым изданием превосходного и эффективного учебного пособия, предназначенного для изучения языка программирования С++ с нуля и удовлетворяющего самым последним стандартам (С++ 11 и С++ 14). Книга задумывалась, с одной стороны, как пособие для тех, кто самостоятельно изучает язык программирования С++, а с другой, она может восприниматься как лекционный курс с проведением практических занятий. Книга содержит полный набор сведений о синтаксисе и концепции языка С++, необходимый для успешного анализа и составления эффективных программных кодов. Материал книги излагается последовательно и сопровождается большим количеством наглядных примеров, разноплановых практических задач и детальным разбором их решений. К каждому разделу прилагается обширный список задач для самостоятельного решения, а также контрольные вопросы (ответы на которые размещены на виртуальный CD).

Книга отличается предельной ясностью, четкостью и доступностью изложения, что вкупе с обширной наглядной практикой (задачами и примерами) позволяет ее рекомендовать как отличный выбор для изучения С++ в соответствии с последними стандартами.

Виртуальный CD с программными кодами, средой разработки программ на С++, примерами, ответами и многими дополнительными материалами доступен для скачивания на сайте www.nit.com.ru

Контактные телефоны издательства:

(812) 412 70 25, (812) 412 70 26, (044) 516 38 66

Официальный сайт: www.nit.com.ru

© Наука и техника (оригинал-макет), 2016

© Васильев А. Н., Прокди, 2016

СОДЕРЖАНИЕ

Часть I. Процедурное программирование в C++ 11

Глава 1. Основы языка C++ 12

ПРОЦЕДУРНОЕ И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ	12
СТРУКТУРА ПРОГРАММЫ В C++	14
СОЗДАНИЕ ПРОСТОЙ ПРОГРАММЫ	14
ИСПОЛЬЗОВАНИЕ ПЕРЕМЕННЫХ	16
ОБЪЯВЛЕНИЕ И ИНИЦИАЛИЗАЦИЯ ПЕРЕМЕННОЙ	18
БАЗОВЫЕ ТИПЫ ДАННЫХ	19
КОНСТАНТЫ И ЛИТЕРАЛЫ	21
АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ	23
ЛОГИЧЕСКИЕ ОПЕРАТОРЫ	27
ОПЕРАТОРЫ СРАВНЕНИЯ	28
ПОБИТОВЫЕ ОПЕРАТОРЫ И ДВОИЧНОЕ ПРЕДСТАВЛЕНИЕ ЧИСЕЛ	28
ОПЕРАТОР ПРИСВАИВАНИЯ И ПРИВЕДЕНИЕ ТИПОВ	32
ТЕРНАРНЫЙ ОПЕРАТОР	35
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	36
Координаты брошенного под углом тела	36
Средняя скорость движения мотоциклиста	37
Высота орбиты спутника	38
Колебания маятника	40
Комплексные числа	41
Прыгающий мячик	43
Умножение на два в степени	45
Решение простого уравнения	45
Атака подводной лодки	47

РЕЗЮМЕ	48
КОНТРОЛЬНЫЕ ВОПРОСЫ	49
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	50

Глава 2. Управляющие инструкции55

УСЛОВНЫЙ ОПЕРАТОР IF()	55
УСЛОВНЫЙ ОПЕРАТОР SWITCH()	59
ОПЕРАТОР ЦИКЛА FOR()	63
ОПЕРАТОР ЦИКЛА WHILE()	70
ИНСТРУКЦИЯ БЕЗУСЛОВНОГО ПЕРЕХОДА.....	73
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	74
Вычисление синуса	74
Вычисление произведения	76
Расчет траектории тела.....	77
Решение уравнения методом последовательных итераций.....	79
Калькулятор	81
Вычисление объема методом Монте-Карло	83
РЕЗЮМЕ	85
КОНТРОЛЬНЫЕ ВОПРОСЫ	86
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	87

Глава 3. Указатели, ссылки и массивы91

ОБЪЯВЛЕНИЕ И ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ	92
АДРЕСНАЯ АРИФМЕТИКА И СРАВНЕНИЕ УКАЗАТЕЛЕЙ	94
МНОГОУРОВНЕВАЯ АДРЕСАЦИЯ	95
ЗНАКОМСТВО СО ССЫЛКАМИ	97
СТАТИЧЕСКИЕ ОДНОМЕРНЫЕ МАССИВЫ.....	100
УКАЗАТЕЛЬ НА МАССИВ	101
ДВУМЕРНЫЕ МАССИВЫ	103
ИНИЦИАЛИЗАЦИЯ МАССИВОВ	106
МАССИВЫ СИМВОЛОВ.....	108
МАССИВЫ УКАЗАТЕЛЕЙ	111

ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	112
Скалярное произведение векторов	113
Векторное произведение	114
Сортировка массива методом пузырька	115
Умножение квадратных матриц	117
Определитель матрицы	118
Математическое ожидание для дискретной случайной величины	119
Метод Ньютона	120
Линейная регрессионная модель	122
РЕЗЮМЕ	125
КОНТРОЛЬНЫЕ ВОПРОСЫ	126
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	127

Глава 4. Функции

ОБЪЯВЛЕНИЕ И ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ	131
МЕХАНИЗМЫ ПЕРЕДАЧИ АРГУМЕНТОВ	135
ПЕРЕДАЧА УКАЗАТЕЛЯ АРГУМЕНТОМ ФУНКЦИИ	138
ПЕРЕДАЧА МАССИВА АРГУМЕНТОМ ФУНКЦИИ	140
ПЕРЕДАЧА СТРОКИ АРГУМЕНТОМ ФУНКЦИИ	143
АРГУМЕНТЫ ФУНКЦИИ MAIN()	144
АРГУМЕНТЫ ПО УМОЛЧАНИЮ	145
ВОЗВРАЩЕНИЕ ФУНКЦИЕЙ УКАЗАТЕЛЯ	147
ВОЗВРАЩЕНИЕ ФУНКЦИЕЙ ССЫЛКИ	148
УКАЗАТЕЛИ НА ФУНКЦИИ	150
РЕКУРСИЯ	152
ПЕРЕГРУЗКА ФУНКЦИЙ	153
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	158
Функция для вычисления гиперболического синуса	158
Вычисление произведения	160
Метод половинного деления	161
Метод хорд	163
Метод Ньютона и перегрузка функции вычисления корня	164

Вывод строки	167
Вычисление статистических характеристик	168
Транспонирование матрицы	170
РЕЗЮМЕ	173
КОНТРОЛЬНЫЕ ВОПРОСЫ	174
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	175

Глава 5. Текстовые строки и динамические массивы

СОЗДАНИЕ И ИНИЦИАЛИЗАЦИЯ СТРОК	178
НУЛЬ-СИМВОЛ ОКОНЧАНИЯ СТРОКИ	182
ФУНКЦИИ ДЛЯ РАБОТЫ СО СТРОКАМИ И СИМВОЛАМИ	188
СТРОЧНЫЕ ЛИТЕРАЛЫ	190
ДВУМЕРНЫЕ СИМВОЛЬНЫЕ МАССИВЫ	191
ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ ПАМЯТИ	193
ДИНАМИЧЕСКИЕ МАССИВЫ	196
МНОГОМЕРНЫЕ ДИНАМИЧЕСКИЕ МАССИВЫ	197
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	201
Раскодировщик	201
Кодировщик	202
Сложение целочисленных матриц	206
Итерационный процесс	207
Вычисление косинуса	209
РЕЗЮМЕ	212
КОНТРОЛЬНЫЕ ВОПРОСЫ	213
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	213

Глава 6. Структуры, объединения и перечисления

СТРУКТУРЫ	217
МАССИВЫ СТРУКТУР	221
ПЕРЕДАЧА СТРУКТУР АРГУМЕНТАМИ ФУНКЦИЙ	224

УКАЗАТЕЛИ НА СТРУКТУРЫ	226
БИТОВЫЕ РАЗМЕРЫ ПОЛЯ	232
ОБЪЕДИНЕНИЯ	233
ПЕРЕЧИСЛЕНИЯ И ОПРЕДЕЛЕНИЕ ТИПОВ	237
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	239
Бинарное дерево	239
Комплексные числа	243
Комплексная экспонента	245
Расстояние между точками	246
Пересечение прямых	247
Корни квадратного уравнения	249
РЕЗЮМЕ	250
КОНТРОЛЬНЫЕ ВОПРОСЫ	252
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	252

Часть II. Объектно-ориентированное программирование в C++

Глава 7. Классы и объекты

ОБЪЯВЛЕНИЕ КЛАССА	260
ОТКРЫТЫЕ И ЗАКРЫТЫЕ ЧЛЕНЫ КЛАССА	266
СТАТИЧЕСКИЕ ЧЛЕНЫ КЛАССА	268
ПЕРЕГРУЗКА МЕТОДОВ	272
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	273
Вычисление логарифма	273
Модуль и аргумент комплексного числа	275
Схема Бернулли	276
Метод последовательных итераций	277
Полет тела	278
РЕЗЮМЕ	280
КОНТРОЛЬНЫЕ ВОПРОСЫ	281
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	282

Глава 8. Работа с объектами	285
ПЕРЕДАЧА ОБЪЕКТОВ АРГУМЕНТАМИ	285
ВОЗВРАЩЕНИЕ РЕЗУЛЬТАТОМ ОБЪЕКТОВ	286
УКАЗАТЕЛИ НА ОБЪЕКТЫ	288
УКАЗАТЕЛИ НА ЧЛЕНЫ КЛАССА	291
ИСПОЛЬЗОВАНИЕ ССЫЛОК НА ОБЪЕКТЫ	294
МАССИВЫ ОБЪЕКТОВ	298
ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ ПАМЯТИ ПОД ОБЪЕКТЫ	300
ДРУЖЕСТВЕННЫЕ ФУНКЦИИ И КЛАССЫ	301
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	306
Комплексная экспонента	306
Динамический список	307
Векторное произведение	310
Интерполяционный полином	311
Производная для полинома	313
Матричная экспонента	314
РЕЗЮМЕ	317
КОНТРОЛЬНЫЕ ВОПРОСЫ	318
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	318

Глава 9. Конструкторы и деструкторы	322
СОЗДАНИЕ И ПЕРЕГРУЗКА КОНСТРУКТОРА	322
ИСПОЛЬЗОВАНИЕ ДЕКТРУКТОРОВ	326
ВЫЗОВ КОНСТРУКТОРА	330
КОНСТРУКТОР СОЗДАНИЯ КОПИИ	333
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	335
Создание бинарного дерева	336
Ряд для экспоненты	338
Поле-объект	339
Поле-массив объектов	341
Вызов в конструкторе метода	343

РЕЗЮМЕ	345
КОНТРОЛЬНЫЕ ВОПРОСЫ	346
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	346

Глава 10. Перегрузка операторов..... 350

ВНЕШНЯЯ ОПЕРАТОРНАЯ ФУНКЦИЯ ДЛЯ ПЕРЕОПРЕДЕЛЕНИЯ ИНАРНОГО ОПЕРАТОРА.....	351
ПЕРЕГРУЗКА ОПЕРАТОРНОЙ ФУНКЦИИ	353
ПЕРЕОПРЕДЕЛЕНИЕ УНАРНЫХ ОПЕРАТОРОВ ВНЕШНИМИ ФУНКЦИЯМИ	356
ПЕРЕГРУЗКА ОПЕРАТОРОВ МЕТОДАМИ КЛАССА	362
ПЕРЕГРУЗКА ОПЕРАТОРА ПРИСВАИВАНИЯ	364
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	365
Индексирование объектов	365
Скалярное и векторное произведение векторов	368
Операции с матрицами	371
Вектор-функция.....	374
Операции с полиномами.....	376
РЕЗЮМЕ	380
КОНТРОЛЬНЫЕ ВОПРОСЫ	381
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	381

Глава 11. Наследование и виртуальные функции 386

НАСЛЕДОВАНИЕ КЛАССОВ И ТИПЫ НАСЛЕДОВАНИЯ	386
ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ И ВИРТУАЛЬНЫЕ ФУНКЦИИ.....	392
МНОГОУРОВНЕВОЕ НАСЛЕДОВАНИЕ	397
МНОГОКРАТНОЕ НАСЛЕДОВАНИЕ	399
КОНСТРУКТОРЫ И ДЕКТРУКТОРЫ ПРИ НАСЛЕДОВАНИИ	402
ЧИСТО ВИРТУАЛЬНЫЕ МЕТОДЫ И АБСТРАКТНЫЕ КЛАССЫ	405
СУЩЕСТВУЮЩИЕ НЕСУЩЕСТВУЮЩИЕ ЧЛЕНЫ КЛАССА	407

ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	409
Комплексные числа	409
Наследование операторных функций	413
Преобразование Фурье	416
Произведение полиномов и ряд Тейлора	421
РЕЗЮМЕ	426
КОНТРОЛЬНЫЕ ВОПРОСЫ	427
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	427

Глава 12. Шаблоны **432**

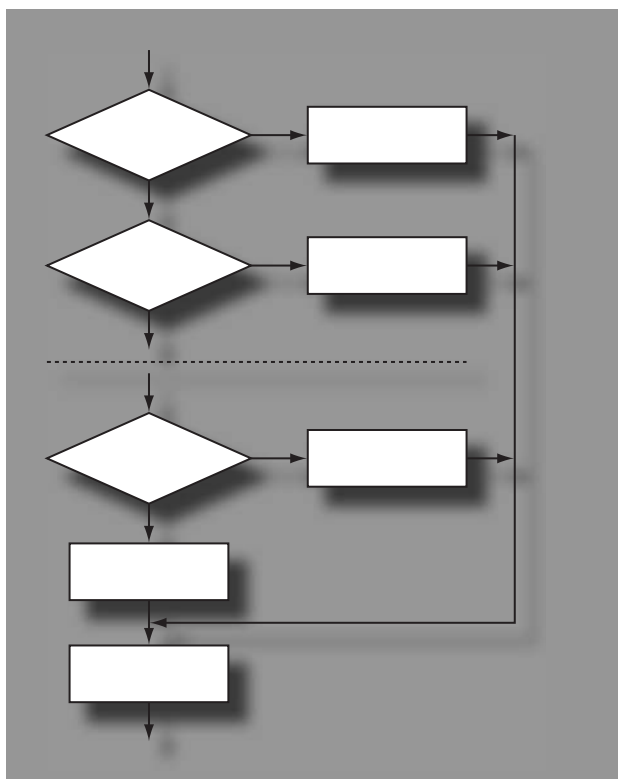
ОБОБЩЕННЫЕ ФУНКЦИИ	432
ПЕРЕГРУЗКА ОБОБЩЕННЫХ ФУНКЦИЙ	436
ОБОБЩЕННЫЕ КЛАССЫ	438
ТИПЫ ПО УМОЛЧАНИЮ И ЯВНАЯ СПЕЦИАЛИЗАЦИЯ КЛАССА	440
ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ	444
Обобщенная экспонента	444
Перегрузка операторов	446
Перестановка элементов массива	448
Поиск совпадений	449
Наследование шаблона	451
Создание обобщенного дерева	452
РЕЗЮМЕ	455
КОНТРОЛЬНЫЕ ВОПРОСЫ	456
ЗАДАЧИ ДЛЯ САМОСТОЯТЕЛЬНОГО РЕШЕНИЯ	457

Часть III. Среды разработки **460**

Глава 13. Компиляторы и интегрированные среды разработки **461**

Глава 14. Работа в Dev-C++ **473**

Процедурное программирование в C++



ЧАСТЬ I

Глава 1

Основы языка C++

Знать много языков – значит иметь много ключей к одному замку.

Вольтер

Среди множества языков программирования C++ занимает особое место. Он достаточно прост, лаконичен и исключительно эффективен. Язык C++ создан профессионалами для профессионалов и является расширением языка C для поддержки объектно-ориентированной парадигмы программирования.

Процедурное и объектно-ориентированное программирование

Преодоленные трудности – это успешно реализованные возможности.

У. Черчилль

Обычно даже очень полезные вещи создаются по необходимости, как следствие разрешения какой-то проблемы. Это же относится и к объектно-ориентированному программированию. Хотя такие языки программирования, как, например, C или Pascal, в свое время и служили мощным инструментом для создания нетривиальных проектов, но в конце концов наступил момент, когда рамки этих (или других аналогичных им) языков программирования стали слишком тесны для постоянно возрастающих потребностей рынка программного обеспечения.

Дело в том, что упомянутые языки реализуют концепцию процедурного программирования. Процедурное программирование подразумевает возможность создания в рамках программы локальных переменных, функций и процедур. Любая программа оперирует данными и содержит код для обработки этих данных. В языках, поддерживающих парадигму процедурного программирования, приоритет остается за кодом для обработки данных. Другими словами, функциональность программы определяется в основном набором процедур и функций для обработки данных. Сами данные при

этом имеют второстепенное значение. Одни и те же процедуры применяют к данным разного рода.

Хотя такой подход имеет право на существование и долгое время был самым прогрессивным, с помощью процедурных языков программирования очень большие проекты создавать сложно. Поэтому на замену процедурному программированию пришло программирование объектно-ориентированное.

Основная идея, положенная в основу объектно-ориентированного подхода, проста и элегантна и состоит в том, чтобы переподчинить код, используемый для обработки данных, этим самым данным. В объектно-ориентированных языках данные играют решающую роль при определении методов обработки. Здесь следует четко понимать, что необходимость перехода к объектно-ориентированному программированию связана в первую очередь с проблемой читабельности программного кода. Если предстоит обработать значительный массив данных и для этого используется большое число процедур и функций, у разработчика или пользователя могут возникнуть (и, как правило, возникают!) проблемы с корректным сопоставлением данных и кода для их обработки. В рамках объектно-ориентированного подхода такое связывание осуществляется на уровне структуры программы и является неотъемлемой частью синтаксиса соответствующего языка программирования (в том числе это относится к C++).

Любой объектно-ориентированный язык программирования базируется на трех механизмах, которые называются инкапсуляцией, полиморфизмом и наследованием.

Под инкапсуляцией подразумевают объединение, связывание в одно целое данных и программного кода для обработки данных. Базовой единицей инкапсуляции является класс, а конкретный экземпляр класса называется объектом. Классы и объекты обсуждаются в последующих главах книги.

Полиморфизм позволяет использовать единый унифицированный интерфейс для выполнения однотипных действий с различными данными. В C++ полиморфизм реализуется через перегрузку функций, методов и операторов. Наконец, наследование позволяет одному объекту получать свойства другого объекта. Это исключительно полезный механизм, который позволяет существенно сокращать объем программного кода, обеспечивать преемственность различных версий программ и лежит в основе принципа классификации объектов. Более подробно все указанные особенности объектно-ориентированного подхода обсуждаются далее в книге на конкретных примерах. На данном этапе читателю следует лишь уяснить, что перечисленными свойствами обладает любой объектно-ориентированный язык программирования, не только C++.

Структура программы в C++

Всякая идея для своего успеха нуждается в жертвах.

Э. Ренан

В общем случае программа C++ состоит из нескольких блоков или частей. С некоторой натяжкой можно утверждать, что таких частей четыре:

1. **Блок заголовков программы.** Обычно в этом блоке с помощью инструкции `#include` подключаются внешние файлы.
2. **Блок с объявлением классов (базовых и производных), прототипами и объявлениями функций.**
3. **Главный метод программы:** каждая программа имеет такой метод. У метода стандартное название `main()`.
4. **Блок с описанием функций** (прототип которых указан во втором блоке).

При этом обязательными являются только первый и третий блоки: программа содержит блок подключения файлов и главный метод `main()`.

Метод – это фактически синоним слова *функция* (или *процедура*). Запуск программы означает выполнение ее главного метода `main()`. У программы может быть один и только один метод `main()`.

Особенность языка C++ состоит еще и в том, что этот язык в определенном смысле переходной – он позволяет создавать как объектно-ориентированные программы, так и программы, подобные тем, что характерны для процедурных языков программирования. Далее рассматриваются наиболее простые программы C++. Программы, в которых реализуются принципы объектно-ориентированного программирования, приводятся во второй части книги.

Создание простой программы

Показная простота – это утонченное лицемерие.

Ф. Ларошфуко

Остановимся на консольных программах, то есть таких, которые предназначены для работы в режиме командной строки. Хотя практическое значение консольного программирования не очень велико, в учебном плане эти программы являются полезными, поскольку позволяют сосредоточиться в первую очередь на особенностях языка программирования.

В общем случае процесс создания программы можно условно разбить на несколько этапов. Сначала нужно набрать программный код. Делать это лучше в специальном редакторе, но в принципе может пригодиться и обычный текстовый редактор. Далее программа компилируется, и в случае успеха она готова к использованию. В листинге 1.1 приведен код программы, в результате выполнения которой в консольной строке отображается сообщение `Hello, World!`.

Листинг 1.1. Программа для отображения приветствия

```
#include <iostream>
using namespace std;
int main()
{
    // Выводится сообщение
    cout <<"Hello, World!\n";
    return 0;
}
```

Чтобы после отображения приветственной фразы выполнялся переход к следующей строке окна консоли, в отображаемый на экране текст добавлена инструкция `\n`. Далее более детально остановимся на анализе программного кода.

Первые две строки в листинге 1.1 формируют раздел заголовков программы. Так, инструкция `#include` используется для присоединения внешнего файла, название которого указывается после инструкции в двойных кавычках или угловых скобках, то есть в формате `#include "имя файла"` или `#include <имя файла>`. Как правило, внешние файлы подключаются для того, чтобы можно было использовать те или иные функции и утилиты. В данной программе подключается файл `iostream` для поддержки системы ввода-вывода. Команда `using namespace std` является инструкцией для компилятора использовать стандартную область имен. Помимо этих инструкций, в зависимости от используемого компилятора приходится добавлять дополнительные. Сведения по поводу дополнительных заголовков необходимо искать в справке соответствующих компиляторов. Особенности работы с наиболее популярными компиляторами описаны в приложениях. Здесь и далее условимся, что если не указано явно, то такая конструкция (имеются в виду первые две строки программы, см. листинг 1.1) является стандартным началом программы.

Непосредственный код программы, который определяет ее функциональность, начинается с инструкции `int main()`. В определенном смысле это стандартная конструкция. Формально речь идет об объявлении функции

`main()`, которая в качестве значения возвращает целое число. Об этом свидетельствует инструкция `int` перед именем функции. Именно в выполнении кода функции `main()` состоит работа программы. Сам код содержится в середине блока, который формируется фигурными скобками `{}`.

Текст, который начинается с двойной косой черты `//`, является комментарием и компилятором полностью игнорируется. Поэтому строка `// Выводится сообщение на функциональности программы не сказывается`. Назначение комментариев – облегчить работу пользователя с кодом программы. Когда программа небольшая, назначение команд и блоков является, как правило, очевидным. Однако при создании существенных по размеру программ наличие поясняющих комментариев становится неотложной необходимостью, особенно если программа создается для использования сторонними пользователями.

В данном варианте программы использовалась команда `cout << "Hello, World!\n"`, которая состоит из ключевого слова-идентификатора `cout` (сокращение от *console output*, означает устройство вывода, в данном случае – экран компьютера), оператора вывода `<<` и текста вывода в двойных кавычках (в данном случае это фраза `"Hello, World!"`). После вывода на экран указанной справа от оператора вывода фразы или текста курсор остается в той же строке. Если бы в программе содержались команды вывода других данных, они бы выводились в той же строке. Чтобы перейти к новой строке, используют инструкцию `\n`, что и было сделано, поэтому соответствующая команда имеет вид `cout << "Hello, World!\n"`. В результате ее выполнения на экране монитора отображается фраза `Hello, World!` и выполняется переход к новой строке.

В общем случае после инструкции `return` указывается значение, возвращаемое функцией в качестве результата. Выше использована команда `return 0`, что в определенном смысле является формальным подтверждением того, что работа программы завершена корректно. Каждая команда заканчивается точкой с запятой – это общее правило (хотя у него есть и исключения).

Использование переменных

Гораздо легче открывать и провозглашать общие принципы, чем применять их.

У. Черчилль

Назначение программы состоит в оперировании данными. Простейшим примером является использование переменной. Как правило, **под перемен-**

ной подразумевают именованную область памяти, к которой можно обратиться в процессе выполнения программы.

В качестве примера рассмотрим программу, в которой пользователь на запрос программы вводит в командной строке свой возраст, после чего программой выводится сообщение о возрасте пользователя. Код этой программы приведен в листинге 1.2.

Листинг 1.2. Программа с переменной

```
#include <iostream>
using namespace std;
int main() {
    int age;                                // Переменная для записи возраста
    cout << "How old are You?\n"; // Сколько Вам лет?
    cin >> age;                            // Нужно указать возраст
    cout << "You are " << age << " years old\n"; // Результат
    return 0;
}
```

В сравнении с предыдущим примером в данном случае имеется несколько существенных отличий. Во-первых, в программе использована переменная `age`. Переменная инициализируется с помощью инструкции `int age`. Ключевое слово `int` указывает на то, что переменная `age` (это имя переменной) принадлежит к целочисленному типу. После инициализации переменную можно использовать в программе.

Сначала программой выводится сообщение `How old are You?`. Далее следует команда `cin >> age`. Она состоит из идентификатора консольного устройства ввода `cin` (сокращение от *console input*), оператора ввода `>>` и имени переменной `age`, которой в качестве значения присваивается число, которое вводит пользователь в командной строке. Следующая команда вывода содержит несколько операторов вывода `<<`, после которых последовательно указываются данные, выводимые на экран. Соответствующая строка формируется тремя фрагментами: текстовым `"You are"`, значением переменной `age`, а также фрагментом `"years old"`. Обращаем внимание читателя на наличие пробелов в конце первого и в начале третьего текстовых фрагментов.

После запуска программы на выполнение на экране появляется сообщение `How old are You?`, а курсор размещен в следующей строке. Там следует ввести целочисленное значение и нажать клавишу `<Enter>`. Если, например, ввести число 33, появляется сообщение `You are 33 years old`.

Объявление и инициализация переменной

Истина часто настолько проста, что в нее не верят.

Ф. Левальд

В общем случае объявление переменной в C++ осуществляется путем указания типа переменной и ее имени. Синтаксис объявления переменной имеет вид `тип имя переменной`. В предыдущем примере целочисленная переменная объявлялась как `int age`. Идентификатором типа является инструкция `int`, а именем переменной `age`. Всего в C++ есть семь базовых типов, и о них речь пойдет несколько позже. Следует понимать: если переменная оглашена как такая, что принадлежит к определенному типу, в дальнейшем изменить ее тип невозможно. Объявляются переменные фактически в любом месте программного кода, однако использовать в программе переменную можно только после того, как ей присвоено значение. В этом случае говорят об **инициализации переменной**. Инициализация переменной также может выполняться в любом месте программы, но *не ранее объявления* этой переменной и *до места первого ее использования*. Объявление и инициализацию переменной часто совмещают. Например, инструкцией `int n=10` объявлена целочисленная переменная `n` со значением 10. Если объявляются несколько переменных одного типа, достаточно перечислить эти переменные, разделенные запятой, после идентификатора типа. То же самое касается инициализации переменных при объявлении. Легитимными с точки зрения синтаксиса C++ являются инструкции, приведенные в листинге 1.3.

Листинг 1.3. Объявление и инициализация переменных

```
// Объявление трех целочисленных переменных n, m и k:
int n, m, k;
// Объявление с одновременной инициализацией нескольких
// переменных:
int one=1, two=2, three, four=4, five;
```

Однако, несмотря на такой демократизм, объявлять переменные лучше в начале соответствующего программного блока, причем желательно указывать персонально для каждой переменной идентификатор типа. Программный код в таком случае пользователем воспринимается намного лучше.

В C++ существует возможность при инициализации переменных использовать не только литералы (явно указанные значения, в соответствии с типом переменной), но и выражения, в которые входят другие переменные. Единственное условие состоит в том, чтобы выражение, на основании которого

инициализируется такая переменная, было легитимным на момент инициализации. В листинге 1.4 приведен код программы, в которой рассчитывается высота, на которую за указанное время поднимается брошенное вверх тело.

Листинг 1.4. Программа для расчета высоты подъема тела

```
#include<iostream>
using namespace std;
int main(){
    // Скорость тела - объявление переменной
    double v;
    // Время полета
    double t=1.2;
    // Ускорение свободного падения
    double g=9.8;
    // Скорость - инициализация переменной
    v=12.3;
    // Высота - динамическая инициализация переменной
    double s=v*t-g*t*t/2;
    cout<<"Height level is "<<s<<"\n";
    return 0;
}
```

Идентификатором действительного типа является инструкция `double`. В начале программы разными способами инициализируются переменные `v`, `g` и `t` (соответственно начальная скорость, ускорение свободного падения и время), после чего с помощью команды `double s=v*t-g*t*t/2` выполняется динамическая инициализация переменной `s`, которая определяет высоту, на которой пребывает тело. Результат расчетов выводится на экран. В выражении инициализации были использованы операторы умножения (`*`), вычитания (`-`) и деления (`/`). В качестве оператора присваивания использован знак равенства (`=`). Про операторы, используемые в C++, речь еще будет идти, а сейчас остановимся на том, какого типа переменные могут использоваться в C++ и в чем их особенность.

Базовые типы данных

*Огласите весь список,
пожалуйста...*

Из к/ф «Операция «Ы» и другие приключения Шурика»

Как уже отмечалось, в C++ существует семь базовых, или основных, типов данных (точнее, семь базовых идентификаторов типа). Они перечислены в таблице 1.1.

Таблица 1.1. Основные типы данных

Идентификатор типа	Тип данных
bool	Логический тип
char	Символьный тип
wchar_t	Символьный двухбайтовый тип
double	Действительные числа двойной точности
float	Действительные числа
int	Целые числа
void	Значение не возвращается

Вместе с идентификаторами типов могут использоваться так называемые модификаторы типа. **Модификаторы типа** – это специальные ключевые слова, которые указываются перед идентификатором типа и позволяют изменять базовый тип. В C++ используются модификаторы `signed` (значения со знаком), `unsigned` (значения без знака), `short` (укороченный тип) и `long` (расширенный тип). Все четыре модификатора могут использоваться для типа `int`. Модификаторы `signed` и `unsigned`, кроме этого, используются с типом `char`, а с типом `double` используют модификатор `long`.

Что касается диапазона значений для данных разных типов, то в разных компиляторах диапазоны значений данных различны. В языке C++ вводятся стандарты только для минимально необходимого диапазона, который должен поддерживать компилятор. В таблице 1.2 перечислены минимальные диапазоны в 32-разрядной среде для данных разных типов с учетом наличия модификаторов типов.

Таблица 1.2. Минимально необходимые диапазоны для данных разных типов

Тип	Количество бит	Диапазон значений
bool	1	Значения <code>true</code> или <code>false</code>
char	8	от -128 до 127
wchar_t	16	от 0 до 65535
double	64	от $2.2\text{E}-308$ до $1.8\text{E}+308$
float	32	от $1.8\text{E}-38$ до $1.8\text{E}+38$
int	32	от -2147483648 до 2147483647
unsigned char	8	от 0 до 255
signed char	8	от -128 до 127
unsigned int	32	от 0 до 4294967295
signed int	32	от -2147483648 до 2147483647

Тип	Количество бит	Диапазон значений
short int	16	от -32768 до 32767
unsigned short int	16	от 0 до 65535
signed short int	16	от -32768 до 32767
long int	32	от -2147483648 до 2147483647
unsigned long int	32	от 0 до 4294967295
signed long int	32	от -2147483648 до 2147483647
double	64	от $2.2\text{E}-308$ до $1.8\text{E}+308$
float	32	от $1.8\text{E}-38$ до $1.8\text{E}+38$
long double	64	от $2.2\text{E}-308$ до $1.8\text{E}+308$

Обращаем внимание читателя на то, что ключевое слово `void` используется при определении функций, которые не возвращают результата. Это функции – аналог процедур в таких языках программирования, как, например, Pascal. Ключевое слово `void` также используется при определении обобщенных указателей.

Константы и литералы

*Нет ничего обманчивее, чем
слишком очевидные факты.*

Артур Конан Дойл

Собственно говоря, в C++ под константами и литералами, как правило, подразумевают одно и то же: это такие значения, предназначенные для восприятия пользователем, которые не могут быть изменены в ходе выполнения программы. Приведенное определение относится большей частью к литералу. **Под константами здесь и далее будем подразумевать именованные ячейки памяти, значения которых фиксируются на начальном этапе выполнения программы и затем в процессе выполнения программы не могут быть изменены.** В этом смысле константы – это те же переменные, но только с фиксированным, определенным единожды, значением.

Для превращения переменной в постоянное запоминающее устройство, то есть в константу, используют идентификатор `const`. Идентификатор указывается перед типом переменной и гарантирует неизменность значения переменной. Само значение переменной присваивается либо при объявлении, либо позже в программе, но только один раз и до первого использования переменной в программе.

Например, инструкцией `const int m=5` инициализируется целочисленная переменная `m` со значением 5. После этого переменную `m` можно использовать в выражениях, однако изменить значение переменной не удастся.

Представление литералов в программном коде существенно зависит от типа, к которому относится литерал. Так, отдельные символы – литералы типа `char` указываются в одинарных кавычках, например `'a'` или `'d'`. Для литералов типа `wchar_t` (напомним, это расширенный 16-битовый символьный тип) перед непосредственно значением указывается префикс `L`. Примеры литералов типа `wchar_t`: `L'a'`, `L'A'`, `L'd'` и `L'D'`.

Несмотря на то, что в C++ нет отдельного типа для строчных (текстовых) переменных, в C++ существуют литералы типа текстовой строки. Соответствующие значения указываются в двойных кавычках. Примером текстового литерала, например, может быть фраза `"Hello, World!"`. Подробнее о работе с текстовыми выражениями рассказывается в главе 5.

Числовые литералы, как и положено, задаются в стандартном виде с помощью арабских цифр. Однако здесь есть один важный момент. Связан он с тем, что в C++ есть несколько числовых типов, поэтому не всегда очевидно, к какому именно числовому типу относится литерал. Здесь действует общее правило: литерал интерпретируется в пределах минимально необходимого типа. Это правило, кстати, относится не только к числовым литералам. Исключение составляет тип `double`: литералы в формате числа с плавающей точкой интерпретируются как значения этого типа (а не типа `float`, как можно было бы ожидать). Тем не менее, в некоторых случаях необходимо в явном виде указать принадлежность литерала к определенному типу. Обычно это делают с помощью специальных суффиксов. Суффикс `F` после числа в формате с плавающей точкой означает принадлежность литерала к типу `float` (например, `5.3F`). Чтобы литерал относился к типу `long double`, используют суффикс `L` (например, `5.3L`). Этот же суффикс, но у целочисленного литерала означает принадлежность последнего к типу `long int` (например, `123L`). Суффикс `U` используется с целочисленными литералами для отнесения их к типу `unsigned int` (например, `123U`).

В C++ также существует возможность работать с восьмеричными и шестнадцатеричными литералами (то есть с целочисленными значениями, записанными в восьмеричной и шестнадцатеричной системах счисления соответственно). Напомним, что в восьмеричной системе счисления число в позиционном представлении записывается с помощью цифр от 0 до 7. Если в восьмеричной системе счисления число записано как $\overline{b_n b_{n-1} \dots b_2 b_1 b_0}$, то в десятичной системе это же число определяется суммой $b_0 8^0 + b_1 8^1 + b_2 8^2 + \dots + b_{n-1} 8^{n-1} + b_n 8^n$. В шестнадцатеричной системе в позиционной записи числа используются

цифры от 0 до 9 и буквы от A (в десятичной системе соответствует числу 10) до F (в десятичной системе соответствует числу 15). По структуре числа $\overline{b_n b_{n-1} \dots b_2 b_1 b_0}$ в шестнадцатеричной системе перевод в десятичную осуществляется как $b_0 16^0 + b_1 16^1 + b_2 16^2 + \dots + b_{n-1} 16^{n-1} + b_n 16^n$. При этом параметры b_n принимают значения в диапазоне от 0 до 15 (символы от A до F заменяются соответствующими числовыми значениями).

В C++ восьмеричные литералы вводятся с помощью арабских цифр, но каждый восьмеричный литерал должен начинаться с нуля. Поэтому хотя с математической точки зрения первый ноль и является незначимым, в C++ литералы 10 и 010 не эквиваленты. Первый является десятичным значением 10, в то время как второй литерал – восьмеричный и в десятичной системе счисления соответствует числу 8. Шестнадцатеричные литералы начинаются с последовательности 0x или 0X. Примеры шестнадцатеричных литералов: 0x10 (десятичное число 16), 0X309 (десятичное число 777), 0x1FA (десятичное число 506).

Арифметические операторы

Нельзя быть настоящим математиком, не будучи немного поэтом.

К. Вейерштрасс

Выражения в C++ содержат, помимо переменных, еще и операторы. Операторы условно можно поделить на арифметические, логические, поразрядные и операторы отношения. Далее описываются особенности работы с каждой из означенных групп операторов. В первую очередь остановимся на арифметических операторах.

Арифметические операторы используются для сложения, вычитания, умножения и деления чисел. Список основных арифметических операторов, которые используются в C++, приведен в таблице 1.3.

Таблица 1.3. Арифметические операторы C++

Оператор	Назначение
+	Сложение
–	Вычитание
*	Умножение
/	Деление. Если операндами являются целые числа, выполняется целочисленное деление

Оператор	Назначение
<code>%</code>	Остаток от деления (деление по модулю)
<code>++</code>	Инкремент
<code>--</code>	Декремент

Первые пять операторов являются бинарными, то есть используются с двумя операндами. За исключением операторов деления по модулю (остаток от целочисленного деления), инкремента и декремента, операторы совпадают с соответствующими математическими операторами. Если оператор деления по модулю особых комментариев не требует, то операторы инкремента и декремента в определенном смысле являются визитной карточкой языка C++. Оператор инкремента даже присутствует в названии языка.

Операторы инкремента и декремента являются унарными (используются с одним операндом). Действие операторов состоит в увеличении или уменьшении значения операнда на единицу. Например, результат выполнения команды `i++` есть увеличение значения переменной `i` на единицу. Другими словами, команды `i++` и `i=i+1` с точки зрения влияния на значение переменной `i` являются эквивалентными. Аналогично в результате команды `i--` значение переменной `i` уменьшается на единицу, как и в результате выполнения команды `i=i-1`.

Операторы инкремента и декремента могут использоваться в префиксной и постфиксной формах. В префиксной форме оператор указывается перед операндом, а в постфиксной форме – после него. Выше операторы инкремента и декремента использовались в постфиксной форме. В префиксной форме соответствующие операции выглядели бы как `++i` и `--i`.

В плане действия на операнд разницы в префиксной и постфиксной формах нет. В результате выполнения команды `++i` значение переменной `i` увеличивается на единицу, как и для команды `i++`. Уменьшение значения переменной `i` на единицу осуществляется при использовании команды `--i`. В этом отношении она не отличается от команды `i--`. Разница между постфиксной и префиксной формами операторов инкремента и декремента проявляется в ситуации, когда эти операторы использованы в выражениях. Естественным образом возникает вопрос относительно переменной-операнда, по отношению к которой применяется операция инкремента или декремента и которая является составной частью более сложного выражения: следует ли сначала вычислить выражение и затем изменить значение переменной или сначала следует изменить значение переменной, а уже потом вычислять выражение? Ответ следующий: для префиксной формы сначала изменяется значение переменной, а затем рассчитывается выражение, а для постфиксной формы выражение вычисляется со старым значением переменной, а только после

этого изменяется значение этой переменной. В этом и состоит разница между префиксной и постфиксной формами операторов инкремента и декремента. Проиллюстрируем сказанное на примере.

В листинге 1.5 приведен код программы, в которой использованы операторы инкремента и декремента в префиксной и постфиксной формах.

Листинг 1.5. Код программы с операторами инкремента и декремента

```
#include<iostream>
using namespace std;
int main(){
    int n,m,i=3,j=3;
    cout<<"At the beginning:\n";
    cout<<"i = "<<i<<"\n";
    cout<<"j = "<<j<<"\n";
    cout<<"After command n=i++ :\n";
    n=i++; // Теперь n=3, а i=4
    cout<<"n = "<<n<<"\n";
    cout<<"i = "<<i<<"\n";
    cout<<" After command m=++j :\n";
    m=++j; // Значение переменных m=4 и j=4
    cout<<"m = "<<m<<"\n";
    cout<<"j = "<<j<<"\n";
    cout<<" After command n=(--i)*(i--) :\n";
    n=(--i)*(i--); // Теперь n=9, а i=2
    cout<<"n = "<<n<<"\n";
    cout<<"i = "<<i<<"\n";
    cout<<" After command m=(--j)*(--j) :\n";
    m=(--j)*(--j); // Теперь m=4, а j=2
    cout<<"m = "<<m<<"\n";
    cout<<"j = "<<j<<"\n";
    cout<<" After command n=(--i)*(i++) :\n";
    n=(--i)*(i++); // Теперь n=1, а i=2
    cout<<"n = "<<n<<"\n";
    cout<<"i = "<<i<<"\n";
    cout<<" After command m=(j--)*(++j) :\n";
    m=(j--)*(++j); // Теперь m=9, а j=2
    cout<<"m = "<<m<<"\n";
    cout<<"j = "<<j<<"\n";
    cout<<" After command n=(--i)*(++i) :\n";
    n=(--i)*(++i); // Теперь n=4, а i=2
    cout<<"n = "<<n<<"\n";
    cout<<"i = "<<i<<"\n";
    return 0;
}
```

В начале программы инициализируются две переменные i и j с одинаковыми начальными значениями, равными 3. Разница между префиксной и постфиксной формами операторов инкремента и декремента иллюстрируется простыми командами: $n=i++$ и $m=++j$. В результате обе переменные i и j увеличивают свое значение на единицу: после выполнения команд они равняются 4. Для переменных n и m результат несколько иной. Переменная n равняется 3, а переменная m равна 4. Причина в том, что при вычислении значения переменной n ей сначала присваивается значение i , а затем переменная i увеличивается на единицу. При вычислении значения переменной m сначала на единицу увеличивается значение переменной j , и только после этого значение переменной j присваивается переменной m .

Однако ситуация не всегда так однозначна. Например, если после означенных выше команд выполнить команду $n=(-i) * (i--)$, получим для переменной i значение 2, а для переменной n значение 9. Алгоритм вычисления значений переменных при этом следующий. Поскольку выражение, на основе которого вычисляется значение переменной n , является произведением двух выражений (то есть $(-i)$ и $(i--)$), то предварительно рассчитываются эти выражения. Результатом выражения $(-i)$ является число 3 (значение i уменьшено на единицу), причем это же значение присваивается переменной i . Такое же значение возвращается выражением $(i--)$, а после присваивания значения переменной n переменная i будет уменьшена еще на единицу.

Если воспользоваться командой $m=(-j) * (--j)$, получим для j значение 2 (дважды значение j уменьшается на единицу), и как результат для m получаем значение 4. Напротив, после выполнения команды $n=(-i) * (i++)$ имеем для i значение 2 и для n значение 1: уменьшается на единицу переменная i (становится равной 1), при данном значении i вычисляется величина n (равна 1), после чего значение i увеличивается на единицу (становится равным 2). Несложно понять, что в результате выполнения команды $m=(j--) * (++j)$ переменная m примет значение 9, а переменная j будет равняться 2. Наконец, команду $n=(-i) * (++i)$ следует понимать так: уменьшаем на единицу переменную i (равняется 1), увеличиваем на единицу переменную i (равняется 2), после чего вычисляем значение переменной n (равняется 4).

Хотя операторы инкремента и декремента во многих отношениях достаточно удобны, использовать их в выражениях следует крайне осторожно.

В C++ существует достаточно удобный формат использования арифметических операторов – так называемая сокращенная форма арифметических операторов. В соответствии с сокращенной формой операторов команды вида `операнд1=операнд1 оператор операнд2` можно записывать в виде `операнд1 оператор=операнд2`. В данном случае `операнд1`

и операнд2 – операнды выражения, а оператор – один из бинарных арифметических операторов. Например, команда `x+=3` эквивалентна команде `x=x+3`, а команду `a=a*b` можно записать как `a*=b`. Сокращенная форма арифметических операторов позволяет существенно упростить код и очень часто используется на практике.

Логические операторы

Опасайтесь ненужных нововведений, особенно если они логически обоснованы.

У. Черчилль

Логические операторы предназначены для работы с операндами логического типа и результатом соответствующих операций являются значения логического типа. В C++ всего три логических оператора, представленных в таблице 1.4.

Таблица 1.4. Логические операторы C++	
Оператор	Назначение
&&	Логическое <i>И</i> . Бинарный оператор. Результатом выражения <code>A&&B</code> является <code>true</code> , если оба операнда <code>A</code> и <code>B</code> равны <code>true</code> . Результатом выражения <code>A&&B</code> является <code>false</code> , если хотя бы один из операндов <code>A</code> или <code>B</code> равен <code>false</code>
	Логическое <i>ИЛИ</i> . Бинарный оператор. Результатом выражения <code>A B</code> является <code>true</code> , если хотя бы один из операндов <code>A</code> или <code>B</code> равен <code>true</code> . Результатом выражения <code>A B</code> является <code>false</code> , если оба операнда <code>A</code> и <code>B</code> равны <code>false</code>
!	Логическое отрицание. Унарный оператор. Результатом выражения <code>!A</code> является значение <code>true</code> , если операнд <code>A</code> равен <code>false</code> . Если операнд <code>A</code> равен <code>true</code> , значение выражения <code>!A</code> равно <code>false</code>

Обращаем внимание читателя на два важных обстоятельства. Во-первых, в языке C++ вместо логических значений можно использовать числовые. При этом имеет место преобразование типов, о котором будет рассказано несколько позже. В соответствии с этим правилом ненулевые значения интерпретируются как `true`, а нулевые значения как `false`. Это исключительно удобный и полезный механизм, который часто используется на практике, особенно в условных операторах.

Во-вторых, в C++ нет логического оператора «*исключающее или*». Обычно эту операцию обозначают как `XOR`. При этом результатом выражения

`A XOR B` является значение `true`, если один и только один из операндов `A` и `B` равен `true`, и `false` в противоположном случае. Однако это обстоятельство не является особенно проблематичным, поскольку с помощью имеющихся в C++ логических операторов можно легко записать эквивалент выражения `A XOR B`. Действительно, легко убедиться, что результатом выражения `(A || B) && ! (A && B)` является `true`, только если операнды `A` и `B` различны (один равен `true`, а другой равен `false`).

Операторы сравнения

Будем наслаждаться своим уделом, не прибегая к сравнениям.

Сенека

Операторы сравнения используются для сравнения значений операндов. Результатом выражения на основе оператора сравнения является логическое значение: `true`, если соответствующее условие выполнено, и `false` в противоположном случае. Операторы сравнения перечислены в таблице 1.5.

Таблица 1.5. Операторы сравнения C++

Оператор	Назначение
<code>></code>	Больше
<code><</code>	Меньше
<code>>=</code>	Больше или равно
<code><=</code>	Меньше или равно
<code>==</code>	Равно
<code>!=</code>	Не равно

Все перечисленные операторы являются бинарными. Примеры использования этих операторов приведены в следующей главе при описании условных операторов.

Побитовые операторы и двоичное представление чисел

Я думаю, что мы получаем одну и ту же информацию, и, возможно, от одних и тех же самых людей.

Н.С. Хрущев

Язык программирования C++ обладает полным набором побитовых операторов. Побитовые операторы применяются при выполнении операций

с битами в двоичном представлении числовых значений. Прежде чем непосредственно рассмотреть сами операторы, кратко остановимся на концепции двоичного представления числовых значений.

Как известно, целые числа представляются в виде последовательности цифр. Такое представление чисел называется позиционным. Весь набор цифр, которые могут использоваться в позиционном представлении числа, определяет систему счисления. В повседневной жизни используется десятичная система счисления, в которой числа представляются цифрами от 0 до 9. В программировании более популярны системы счисления с количеством цифр, равным степени двойки: восьмеричная и шестнадцатеричная. Однако двоичная система счисления – вне конкуренции. В этой системе счисления числа записываются последовательностью из двух цифр: 0 и 1. Каждая позиция в двоичном представлении числа соответствует биту. Таким образом, с помощью бита можно задать два значения: 0 или 1. Если для представления числа используется n бит, то в этом случае существует 2^n различных комбинаций, каждая из которых соответствует отдельному числу. Например, с помощью 8 бит (1 байт) можем записать $2^8 = 256$ чисел.

При представлении двоичным кодом положительных чисел можно было бы использовать стандартное математическое представление числа в двоичной системе. Однако на практике приходится иметь дело и с отрицательными числами, причем с технической точки зрения знаком *минус* здесь не обойтись – минус можно написать на бумаге, а реализовать его в памяти компьютера намного сложнее.

Для определения знака числа используют старший бит в позиционной записи. Нулевой старший бит соответствует положительному числу, а единичный старший бит соответствует отрицательному числу. При этом перевод для положительных чисел из двоичной системы счисления в десятичную осуществляется стандартными методами: если в двоичном представлении число позиционно задается как $\overline{b_n b_{n-1} \dots b_2 b_1 b_0}$ (причем цифры b_i могут принимать значения 0 или 1, а старший бит для положительных чисел $b_n = 0$), то в десятичной системе число вычисляется как $b_0 2^0 + b_1 2^1 + b_2 2^2 + \dots + b_{n-1} 2^{n-1} + b_n 2^n$. С отрицательными числами дела обстоят несколько сложнее. Чтобы перевести отрицательное число с позиционным представлением в двоичной системе $\overline{b_n b_{n-1} \dots b_2 b_1 b_0}$ (старший бит для отрицательного числа $b_n = 1$), необходимо проделать несложную процедуру из двух этапов. Во-первых, производится побитовое инвертирование кода, т.е. каждый бит в представлении числа меняется на противоположный: 0 на 1 и 1 на 0. Во-вторых, результат переводится в десятичную систему и к нему добавляется 1. Это модуль отрицательного числа. Чтобы получить само число, необходимо умножить на -1 . Чтобы перевести отрицательное число из десятичной системы в двоичную, проде-

лывают обратную процедуру: от модуля отрицательного числа отнимается 1, результат переводится в бинарный код, после чего производится побитовое инвертирование. Проиллюстрируем это на примере. Рассмотрим 8-битовое бинарное положительное число 01001011, что в десятичной системе счисления соответствует числу $2^0 + 2^1 + 2^3 + 2^6 = 75$. Определим бинарное машинное представление для отрицательного числа -75 . Отнимаем от модуля числа единицу, получаем 74. Бинарное представление для этого числа 01001010 ($74 = 2^1 + 2^3 + 2^6$). После побитового инвертирования из числа 01001010 получаем 10110101. Это и есть представление числа -75 . В том, что это так, легко убедиться: сложим числа 01001011 и 10110101. Формально получаем 100000000, однако поскольку числа 8-битовые, лишний единичный старший бит отбрасывается, и получаем представление 00000000, что соответствует нулю, как и должно быть.

Теперь рассмотрим основные побитовые операции и операторы, которые используются для этого в языке программирования C++. Список побитовых операторов приведен в таблице 1.6.

Таблица 1.6. Побитовые операторы C++

Оператор	Назначение
&	Побитовое <i>И</i> . Бинарный оператор. Результатом выражения $a \& b$ является число, каждый бит которого в двоичном представлении равен результату сравнения соответствующих битов чисел a и b : значение бита равно 1, если оба сравниваемых бита равны 1. В противном случае значение бита равно 0
	Побитовое <i>ИЛИ</i> . Бинарный оператор. Результатом выражения $a b$ является число, каждый бит которого в двоичном представлении равен результату сравнения соответствующих битов чисел a и b : значение бита равно 1, если хотя бы один из сравниваемых битов равен 1. В противном случае значение бита равно 0
^	Побитовое исключающее <i>ИЛИ</i> . Бинарный оператор. Результатом выражения $a \wedge b$ является число, каждый бит которого в двоичном представлении равен результату сравнения соответствующих битов чисел a и b : значение бита равно 1, если один и только один из сравниваемых битов равен 1. В противном случае значение бита равно 0
~	Побитовое отрицание (дополнение до единицы). Унарный оператор. Результатом выражения $\sim a$ является число, которое получается побитовым инвертированием числа a
>>	Сдвиг вправо. Бинарный оператор. В двоичном представлении числа, указанном слева от оператора, выполняется сдвиг всех битов вправо на число позиций, указанных справа от оператора. При этом старший бит знака остается неизменным, а выходящие за диапазон младшие биты теряются

Оператор	Назначение
<<	Сдвиг влево. Бинарный оператор. В двоичном представлении числа, указанном слева от оператора, выполняется сдвиг всех битов влево на число позиций, указанных справа от оператора, с заполнением младших битов нулями и потерей старших битов

Приведем некоторые примеры использования побитовых операторов. Они представлены в таблице 1.7. Даже в самых простых случаях результат может оказаться несколько неожиданным.

Таблица 1.7. Примеры использования побитовых операторов

Выражение	Значение	Пояснение
5&3	1	В двоичном представлении число 5 имеет вид 101, а число 3 представляется как 011. Побитовое сравнение чисел 101 и 011 с помощью оператора <i>побитового И</i> & дает 001, что в десятичной системе соответствует числу 1
5 3	7	Применение оператора <i>побитового ИЛИ</i> для сравнения чисел 101 и 011 дает 111, что в десятичной системе соответствует числу 7
5^3	6	Применение оператора <i>побитового исключающего ИЛИ</i> ^ для сравнения чисел 101 и 011 дает 110, что в десятичной системе соответствует числу 6
~5	-6	После применения операции <i>побитового инвертирования</i> ~ к числу 5 требует особых пояснений. На самом деле в 8-битовом представлении число 5 имеет вид 00000101. В предыдущих случаях нулевые старшие разряды роли не играли, поэтому они явно не указывались. При инвертировании наличие старших нулевых битов важно. Инвертирование дает 11111010. Это не что иное, как представление в двоичном машинном коде числа -6. Последнее читатель может проверить самостоятельно
5>>2	1	После сдвига вправо на две позиции для числа 5 (двоичный код 101) получаем 001. В десятичной системе это число 1
5<<2	20	После сдвига влево на две позиции для числа 5 (двоичный код 101) получаем 10100. В десятичной системе это число 20

Обращаем внимание читателя на особенности применения операции побитового сдвига к отрицательным числам. Например, результатом выражения -6>>5 является число -1. Дело в том, что в бинарном коде 11111010 для

числа -6 при сдвиге вправо на 5 позиций при условии сохранения значения старшего бита знака получаем код 11111111. Это код числа -1 .

Особенности операций в двоичной системе таковы, что сдвиг в побитовом представлении числа на одну позицию влево означает умножение этого числа на 2. Следует только помнить, что с определенного момента при сдвиге вправо теряются старшие биты. Представим, что число задается 8 битами. Если воспользоваться командой `1<<6`, получим в качестве результата значение $2^6 = 64$. Действительно, десятичное число 1 в двоичной системе в 8-битовом представлении задается как 00000001. После сдвига влево на 6 позиций получаем 01000000, что в десятичной системе соответствует числу 64. Однако если воспользоваться командой `1<<7`, получим в качестве результата -128 . Объясняется это следующим обстоятельством. После сдвига влево на 7 позиций из числа 00000001 получаем число 10000000. Это отрицательное число, о чем свидетельствует старший единичный бит. Переводя это число в десятичную систему, сначала инвертируем бинарный код и получаем 01111111. Это код числа 127. Чтобы получить конечное значение, необходимо прибавить к этому результату 1 и добавить минус $-$ в результате приходим к значению -128 .

Оператор присваивания и приведение типов

Когда мне понадобится ваше мнение, я вам его сообщу.

Линдон Джонсон

Оператор присваивания ранее уже использовался. Здесь остановимся на некоторых особенностях этого оператора. Главная особенность оператора присваивания в C++ состоит в том, что он возвращает значение. Это означает, что выражение с оператором присваивания может, в свою очередь, быть частью другого выражения. Синтаксис использования оператора присваивания таков: переменная=выражение. При этом переменной в качестве значения присваивается результат, возвращаемый при вычислении выражения. Результат выражения является тем значением, которое возвращает оператор присваивания. Поэтому помимо обычных присваиваний вида `n=5` допускается многократное использование оператора присваивания в выражениях. Например, вполне законной является инструкция вида `x=y=z=3`, в результате выполнения которой переменным `x`, `y` и `z` присваивается значение 3. Инструкции могут быть и более замысловатыми. В листинге 1.6 целочисленным переменным `n` и `m` командой `n=(m=6)+3` присваиваются значения 9 и 6 соответственно.

Листинг 1.6. Оператор присваивания

```
#include <iostream>
using namespace std;
int main()
{
    int n,m;
    n=(m=6)+3;
    cout<<m<<"\n";
    cout<<n<<"\n";
    return 0;
}
```

В соответствии с приведенной командой сначала вычисляется выражение $m=6$, значением которого является 6, и при этом такое же значение присваивается переменной m . Далее к этому значению прибавляется число 3 и результат присваивается переменной n .

Еще одна особенность оператора присваивания связана с преобразованием типа в выражениях, содержащих этот оператор. Если в инструкции вида `переменная=выражение` тип результата, возвращаемого выражением, не совпадает с типом переменной, имеет место автоматическое преобразование типов. При этом тип значения выражения приводится в соответствие с типом переменной. Результат такой операции существенно зависит от того, из какого в какой тип данных осуществляется преобразование. В некоторых случаях может иметь место потеря данных: например, если переменной типа `int` присваивается значение переменной типа `double`, как в листинге 1.7.

Листинг 1.7. Автоматическое преобразование типов при присваивании значения

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    double x=3.5;
    a=x;
    cout<<"x="<<x<<"\n";
    cout<<"a="<<a<<"\n";
    return 0;
}
```

В программе объявляется целочисленная переменная `a` типа `int` и инициализируется переменная `x` типа `double` со значением 3.5. Далее в программе переменной `a` в качестве значения присваивается переменная `x`. В дан-

ном случае речь идет о том, что переменной типа `int` присваивается значение типа `double`. Поскольку диапазон значений для типа `double` шире диапазона значений для типа `int`, происходит урезание значения. В частности, при присваивании отбрасывается дробная часть числа, и переменная `a` получает значение 3. Если бы переменной типа `double` присваивалось значение типа `int`, таких проблем не возникало бы.

Преобразование типов может проявляться и совершенно по-иному. В листинге 1.8 приведен пример кода программы, в которой переменной типа `char` присваивается значение типа `int`.

Листинг 1.8. Преобразование `int`-типа в `char`-тип

```
#include <iostream>
using namespace std;
int main() {
    char x;
    int n=68;
    int z;
    x=n;
    z='D';
    cout<<"x="<<x<<"\n";
    cout<<"z="<<z<<"\n";
    return 0;
}
```

Целочисленной переменной `n` типа `int` присваивается значение 68. Далее эта переменная присваивается в качестве значения переменной `x` `char`-типа. В результате автоматического преобразования типов переменная `x`, в соответствии с кодовой таблицей, получает символьное значение `D`. В этом же программном коде проиллюстрирована обратная процедура: командой `z='D'` целочисленной переменной `z` присваивается в качестве значения символ `D`. В результате автоматического преобразования символов этой переменной на самом деле будет присвоен целочисленный код символа в кодовой таблице (значение 68).

Автоматическое приведение типов в некоторых случаях может проявляться достаточно неожиданным образом. Например, если операндами при делении являются целые числа, выполняется целочисленное деление. Допустим, в программе объявлены переменные `int n=13, m=5`. Результатом выражения `n/m` является число 2 (целая часть от деления 13 на 5). Для явного приведения типа выражения тот тип, к которому следует привести выражение, указывается в круглых скобках перед выражением. Чтобы в приведенном примере деление было обычным, не целочисленным, можно воспользоваться командой `(double) n/m`. В результате получим 2.6.

Тернарный оператор

«Иметь двух жен для мужчины неприлично» – сказал турецкий паша, у которого их было три.

Из к/ф «Сильва»

В C++ есть тернарный оператор (у оператора три операнда), который позволяет в зависимости от некоторого условия (первый операнд) выполнять различные действия (второй и третий операнды). Синтаксис вызова оператора таков: условие?выражение1:выражение2. Фактически тернарный оператор представляет собой сокращенную форму условного оператора (об условных операторах речь идет в следующей главе). Пример вызова тернарного оператора приведен в листинге 1.9.

Листинг 1.9. Тернарный оператор

```
#include <iostream>
using namespace std;
int main() {
    int n;
    double x;
    cout<<"Enter n = ";
    cin>>n;
    x=n>0?5.4:3.2;
    cout<<"x = "<<x<<"\n";
    return 0;
}
```

В программе объявляются две переменные: целочисленная переменная `n` типа `int` и переменная `x` типа `double`. Значение переменной `n` вводится с клавиатуры пользователем, а значение переменной `x` определяется с помощью команды `x=n>0?5.4:3.2`. Дело в том, что тернарный оператор возвращает значение. Сначала проверяется указанное первым операндом условие. Если условие выполнено, вычисляется выражение1 после вопросительного знака. Если условие не выполнено, вычисляется выражение2 после двоеточия. Тернарным оператором в качестве значения возвращается значение вычисленного выражения.

В программе командой `n>0?5.4:3.2` проверяется условие `n>0`, и если это так, возвращается значение 5.4. В противном случае возвращается значение 3.2.

На практике в тернарных операторах используются более сложные конструкции, чем просто возвращение в качестве значения числа. Обычно при-

бегают к вызову в тернарных операторах функций, что позволяет не просто возвращать значение в зависимости от проверяемого условия, но и выполнять целые последовательности действий.

Примеры решения задач

Нет ничего более раздражающего, чем хороший пример.

Марк Твен

Рассмотрим некоторые достаточно простые задачи, которые могут решаться в C++ с применением минимальных программных ресурсов. В этих программах создаются и используются константы и переменные, а также применяются операторы ввода-вывода. Задачи предназначены для закрепления изложенного в главе материала.

■ Координаты брошенного под углом тела

Сначала рассмотрим программу, в которой вычисляются координаты тела, брошенного под углом к горизонту. Напомним, что если телу в начальный момент сообщена скорость v и тело брошено под углом α к горизонту, то зависимость x -координаты от времени t дается соотношением $x(t) = v \cos(\alpha)t$. Для y -координаты закон движения имеет вид $y(t) = v \sin(\alpha)t - gt^2/2$, где $g \approx 9.8$ м/с² есть ускорение свободного падения. Время полета тела до падения составляет $T = 2v \sin(\alpha)/g$. В программе вводится значение скорости тела и угол, под которым тело брошено к горизонту. Угол вводится в градусах, поэтому вводимое значение переводится в радианы (умножается на $\pi \approx 3.1415$ и делится на 180). Далее вычисляется время полета тела, и пользователю предлагается ввести момент времени (не превышающий время полета тела), для которого необходимо рассчитать координаты тела. Эти координаты выводятся на экран. Код программы приведен в листинге 1.10.

Листинг 1.10. Тело брошено под углом к горизонту

```
#include<iostream>
#include <cmath>
using namespace std;
int main(){
    //Константа - ускорение свободного падения и число pi:
    const double g=9.8;
    const double pi=3.1415;
```



```
//Начальные и расчетные параметры задачи
//(скорость, угол и время полета):
double v,alpha,T;
//Момент времени и координаты:
double t,x,y;
//Ввод параметров:
cout<<"Enter speed v = ";
cin>>v;
cout<<"Enter angel alpha = ";
cin>>alpha;
alpha=alpha*pi/180;
T=2*v*sin(alpha)/g;
cout<<"Enter time t < "<<T<<": ";
cin>>t;
x=v*t*cos(alpha);
y=v*t*sin(alpha)-g*t*t/2;
cout<<"x = "<<x<<"\n";
cout<<"y = "<<y<<"\n";
return 0;
}
```

Для использования встроенных математических функций в блок заголовков включена команда `#include <cmath>`. Результат выполнения программы может выглядеть следующим образом (жирным шрифтом выделены данные, вводимые пользователем):

```
Enter speed v = 10
Enter angel alpha = 30
Enter time t < 1.02038: 0.5
x = 4.33017
y = 1.27493
```

В программах такого типа разумнее предусматривать возможность вычисления координат для разных моментов времени. Делается это с помощью условных операторов и операторов цикла, о которых речь пойдет в следующих главах.

■ Средняя скорость движения мотоциклиста

Решим задачу о вычислении средней скорости движения мотоциклиста на участке от пункта А до В через пункт Б, если расстояние между пунктами А и Б составляет S_1 , а расстояние между пунктами Б и В равно S_2 . Время движения мотоциклиста между пунктами А и Б равно t_1 , а время движения между пунктами Б и В равно t_2 . Средняя скорость определяется как $V = (S_1 + S_2)/(t_1 + t_2)$. Параметры S_1 , S_2 , t_1 и t_2 вводятся пользователем с клавиатуры. Программный код приведен в листинге 1.11.

Листинг 1.11. Средняя скорость мотоциклиста

```

#include<iostream>
using namespace std;
int main(){
    //Параметры задачи:
    double S1,S2,t1,t2,V;
    //Ввод параметров:
    cout<<"Enter S1 = ";
    cin>>S1;
    cout<<"Enter S2 = ";
    cin>>S2;
    cout<<"Enter t1 = ";
    cin>>t1;
    cout<<"Enter t2 = ";
    cin>>t2;
    V=(S1+S2)/(t1+t2);
    cout<<"Speed V = "<<V<<"\n";
    return 0;
}

```

Результат выполнения программы может выглядеть следующим образом (жирным шрифтом выделен ввод пользователя):

```

Enter S1 = 60
Enter S2 = 100
Enter t1 = 1.5
Enter t2 = 2
Speed V = 45.7143

```

Выше предполагалось, что расстояние вводится в километрах, а время – в часах.

■ Высота орбиты спутника

Следующая задача, которую рассмотрим, состоит в определении высоты орбиты спутника h над поверхностью Земли, если известны масса $M \approx 5.96 \times 10^{24}$ (кг) и радиус $R \approx 6.37 \times 10^6$ (м) Земли, масса спутника m , период его обращения T . Масса спутника в данном случае при расчете высоты орбиты не нужна, а период обращения вводится пользователем. При решении этой задачи воспользуемся тем, что сила гравитационного притяжения между Землей и спутником равна $F = G \frac{mM}{(R+h)^2}$, где $G \approx 6.672 \times 10^{-11}$ (Нм²/кг²) – универсальная гравитационная постоянная. С другой стороны, эту же силу по второму закону Ньютона можно

записать как $F = ma$, где $a = \omega^2(R + h)$ есть центростремительное ускорение, а частота ω связана с периодом T соотношением $\omega = 2\pi/T$. Из этих соотношений получаем $4\pi^2 m(R + h)/T^2 = GmM/(R + h)^2$, что дает $h = \sqrt[3]{\frac{GMT^2}{4\pi^2}} - R$. Соответствующий программный код приведен в листинге 1.12.

Листинг 1.12. Высота орбиты спутника

```
#include<iostream>
#include <cmath>
using namespace std;
int main(){
    //Гравитационная постоянная:
    const double G=6.672E-11;
    //Масса Земли:
    const double M=5.96E24;
    //Радиус Земли:
    const double R=6.37E6;
    //Число pi:
    const double pi=3.1415;
    //Период и высота орбиты:
    double T,h;
    //Ввод периода (в часах):
    cout<<"Enter T = ";
    cin>>T;
    //Перевод часов в секунды:
    T=T*3600;
    //Определение высоты:
    h=pow(G*M*T*T/4/pi/pi, (double)1/3) -R;
    //Перевод в километры:
    h=h/1000;
    cout<<"Height h ="<<h<<" km\n";
    return 0;
}
```

Результат выполнения программы может выглядеть следующим образом (жирным шрифтом выделен ввод пользователя):

```
Enter T = 1,5
Height h = 244.401 km
```

В программе использована встроенная функция `pow()` для вычисления кубического корня. Первым аргументом функции указывается возводимое в степень выражение, второй ее аргумент – степень, в которую возводится

выражение. В данном случае степень равна $1/3$. Однако в силу автоматического преобразования типов при вычислении выражения $1/3$ используется целочисленное деление, в результате чего получаем 0. Чтобы избежать такой неприятности, во втором аргументе функции `pow()` использована инструкция `(double)` для выполнения явного приведения типов.

Обращаем также внимание читателя на способ ввода больших чисел: они вводятся в формате мантиссы и показателя степени. Например, число (литерал) 6.672×10^{-11} вводится как `6.672E-11`, а число 5.96×10^{24} – как `5.96E24`.

■ Колебания маятника

Рассмотрим следующую задачу. Маятник совершает колебания по закону $x(t) = A \sin(\omega t + \varphi_0)$. Частота колебаний ω известна. Известно также, что в начальный момент координата маятника положительна и в k раз меньше амплитуды A , а в момент времени t_1 значение координаты маятника равно A_1 . Напишем программу, в которой определяется амплитуда колебаний A .

Амплитуда колебаний определяется из соотношений $A/k = A \sin(\varphi_0)$ (что дает $\sin(\varphi_0) = 1/k$) и $A_1 = A \sin(\omega t_1 + \varphi_0)$. Можно найти точное аналитическое решение, но в данном случае в этом необходимости нет. Сначала по формуле $\varphi_0 = \arcsin(1/k)$ вычисляем начальную фазу φ_0 , а затем по формуле $A = A_1 / \sin(\omega t_1 + \varphi_0)$ вычисляем амплитуду A . Программный код приведен в листинге 1.13.

Листинг 1.13. Амплитуда колебаний маятника

```
#include<iostream>
#include <cmath>
using namespace std;
int main(){
    //Частота колебаний:
    double omega=0.2;
    //Параметры задачи:
    double A1,t1,k;
    //Амплитуда и начальная фаза:
    double A,phi0;
    //Ввод параметров:
    cout<<"Enter t1 = ";
    cin>>t1;
    cout<<"Enter A1 = ";
    cin>>A1;
```

```

cout<<"Enter k = ";
cin>>k;
phi0=asin(1/k);
A=A1/sin(omega*t1+phi0);
cout<<"Amplitude A = "<<A<<"\n";
return 0;
}

```

В программе для вычисления арксинуса использована встроенная функция `asin()`. Результат выполнения программы может иметь вид:

```

Enter t1 = 10
Enter A1 = 20
Enter k = 4
Amplitude A = 25.7604

```

Жирным шрифтом, как и ранее, выделены вводимые пользователем значения.

■ Комплексные числа

Напомним, что любое комплексное число может быть представлено в виде $z = x + iy$, где x – действительная часть комплексного числа, y – мнимая часть комплексного числа, i – мнимая единица ($i^2 = -1$). Такое представление комплексного числа называется алгебраическим. Существует тригонометрическое представление $z = |z| \exp(i\varphi)$, где модуль комплексного числа $|z| = \sqrt{x^2 + y^2}$, а аргумент φ такой, что $|z| \cos(\varphi) = x$ и $|z| \sin(\varphi) = y$.

Напишем программу, которой вычисляется целочисленная степень комплексного числа, т.е. значение выражения z^n . При этом воспользуемся соотношением $z^n = |z|^n \exp(in\varphi) = |z|^n \cos(n\varphi) + i |z|^n \sin(n\varphi)$. Таким образом, действительной частью числа z^n является $|z|^n \cos(n\varphi)$, а комплексной – $|z|^n \sin(n\varphi)$. Программный код, в котором реализовано вычисление степени комплексного числа, приведен в листинге 1.14.

Листинг 1.14. Степень комплексного числа

```

#include<iostream>
#include <cmath>
using namespace std;
int main(){
    //Действительная и мнимая часть:
    double x,y,X,Y;
    //Модуль и аргумент:
    double r,phi,R,Phi;

```

```

//Показатель степени:
int n;
//Ввод параметров:
cout<<"Real part x = ";
cin>>x;
cout<<"Imaginary part y = ";
cin>>y;
cout<<"Power n = ";
cin>>n;
//Вычисление результата:
phi=atan2(y,x);
r=sqrt(x*x+y*y);
R=pow(r,n);
Phi=n*phi;
X=R*cos(Phi);
Y=R*sin(Phi);
cout<<"The result is:\n";
cout<<"Re-part "<<X<<"\n";
cout<<"Im-part "<<Y<<"\n";
return 0;
}

```

В результате выполнения программы получаем, например, следующее (жирным шрифтом выделены вводимые пользователем значения):

```

Real part x = 1
Imaginary part y = -2
Power n = 3
The result is:
Re-part -11
Im-part 2

```

В программе вводится действительная и мнимая части комплексного числа, а также целочисленная степень, в которую нужно это число возвести. От алгебраического представления числа переходим к тригонометрическому, для чего вычисляется модуль числа и аргумент. Аргумент вычисляется с помощью встроенной функции `atan2()`, аргументами которой передаются комплексная и действительная части комплексного числа, а результатом является аргумент (угол на соответствующую точку на плоскости). Корень квадратный при вычислении модуля вычисляется с помощью встроенной функции `sqrt()`.

Далее вычисляется модуль и аргумент комплексного числа, которое получается возведением в указанную степень исходного комплексного числа. По модулю и аргументу вычисляется действительная и мнимая части, которые и выводятся на экран.

■ Прыгающий мячик

В рассмотренных выше задачах использовались элементарные арифметические операции. На практике в таких простых случаях применение программных средств является малоэффективным. Программы обычно создаются для того, чтобы решать сложные вычислительные задачи, с использованием алгоритмов, основанных на циклах, и проверкой всевозможных условий. Далее рассмотрим задачу, в которой для реализации точки ветвления используется тернарный оператор. Условие задачи формулируется следующим образом.

Мяч бросают без начальной скорости с высоты h . Мяч, долетая до пола, отбивается (без потери энергии) и подпрыгивает вертикально вверх, затем снова падает на пол, отбивается и т.д. Надо написать программу, которой определяется высота мячика над полом в заданный пользователем момент времени.

Программный код, которым решается эта задача, приведен в листинге 1.15.

Листинг 1.15. Прыгающий мячик

```
#include<iostream>
#include <cmath>
using namespace std;
int main(){
    const double g=9.8;
    //Высота:
    double h=78.4;
    //Полупериод, вводимое пользователем время и координата:
    double T,t,x;
    //Количество полупериодов:
    int n;
    //Ввод пользователем момента времени:
    cout<<"Enter t = ";
    cin>>t;
    T=sqrt(2*h/g);
    n=(int)t/T;
    t=n%2?T-(t-n*T):t-n*T;
    x=h-g*t*t/2;
    cout<<"x = "<<x<<" m"<<endl;
    return 0;
}
```

Желающие могут самостоятельно протестировать работу программы. Параметры подобраны так, что до пола мячик долетает за 4 секунды. Такое же время уходит на подъем на максимальную высоту.

При составлении программного кода принята во внимание симметрия задачи. Совершенно очевидно, что для восстановления таких параметров мячика, как высота и скорость, в произвольный момент времени достаточно проследить динамику мячика от начала падения до отбивания от пола. После отбивания от пола до подъема на максимальную высоту (которая совпадает с начальной высотой, с которой бросают мячик) динамика мячика может быть восстановлена, если пустить время в обратном направлении: при отражении от пола мгновенно (такое используется приближение) меняется направление скорости (при этом модуль скорости не меняется). Поэтому, прокрутив картинку в обратном порядке, получим динамику мячика после отбивания от пола. В частности, если при падении мячик пребывал на некоторой высоте в момент времени t (время t отсчитывается от момента бросания мячика), то при движении вверх на этой же высоте мячик будет в момент времени $T - t$ (здесь время t отсчитывается от момента отбивания мячика от пола, а через T обозначено время падения мячика).

В программе реализуется следующий алгоритм. Пользователем вводится время (переменная t), для которого необходимо рассчитать положение мячика. Затем в соответствии с формулой $T = \sqrt{2h/g}$ командой $T = \text{sqrt}(2 * h / g)$ вычисляется время падения мячика до первого отбивания от пола. Время T – это фактически полупериод движения мячика, поскольку за время $2T$ мячик оказывается в той же точке с той же скоростью.

Целочисленный параметр n , вычисляемый командой $n = (\text{int}) t / T$, является целой частью от деления времени t на полупериод T . Если n – число четное, то в момент времени $t - n * T$ мячик был в том же месте, что и в момент времени t . Однако момент времени $t - n * T$ однозначно попадает в интервал времени, когда мячик падал с начальной высоты до первого удара о пол. Если n – число нечетное, то мячик будет находиться в том же месте, где он находился в момент времени $T - (t - n * T)$. Поэтому в программе переменная t переопределяется с помощью команды $t = n \% 2 ? T - (t - n * T) : t - n * T$. Переменной t в качестве значения присваивается результат вычисления тернарного оператора. Проверяемым является условие $n \% 2$ – остаток от деления n на 2. Это число, равное 0 или 1. Напомним, что нулевое значение интерпретируется как логическое значение `false`, а прочие числовые значения (в том числе и 1) – как `true`. Если остаток от деления равен 1 (n – число нечетное), переменной t в качестве значения присваивается результат выражения $T - (t - n * T)$. В противном случае переменной t присваивается значение $t - n * T$.

После выполнения тернарного оператора значение переменной t попадает во временной интервал, когда мячик падал первый раз. Высота шарика над полом вычисляется в соответствии с формулой $x = h - gt^2/2$ командой $x = h - g * t * t / 2$.

■ Умножение на два в степени

Рассмотрим простую программу, представленную в листинге 1.16, в которой введенное пользователем целое число умножается на 2 в целочисленной степени (показатель степени также вводится пользователем). Для выполнения умножения использован оператор побитового сдвига.

Листинг 1.16. Умножение на два в степени

```
#include<iostream>
using namespace std;
int main(){
    //Число, показатель степени и результат:
    int number,n,result;
    //Ввод пользователем числа:
    cout<<"Enter number = ";
    cin>>number;
    //Ввод пользователем показателя степени:
    cout<<"Enter n = ";
    cin>>n;
    //Результат умножения:
    result=number<<n;
    cout<<"result = "<<result<<endl;
    return 0;
}
```

Результат выполнения программы может выглядеть, например, так (жирным шрифтом выделен ввод пользователя):

```
Enter number = 3
Enter n = 4
result = 48
```

Для понимания работы программы нужно учесть, что побитовый сдвиг на одну позицию эквивалентен умножению соответствующего числа на 2. Поэтому сдвиг, например, на 4 позиции означает умножение числа на 16 ($2^4 = 16$).

■ Решение простого уравнения

В качестве небольшой иллюстрации применения логических операторов приведем программу, которой относительно переменной x решается уравнение вида $ax = b$. Особой интриги в этом уравнении нет. При ненулевых значениях a и b решением является $x = b/a$. Если $a = 0$, то возможны два варианта: при $b = 0$ решением является любое число x , а при $b \neq 0$

уравнение решений не имеет. Именно эти варианты и необходимо предусмотреть в программе. Сразу отметим, что приведенный в листинге 1.17 код не является оптимальным, это всего лишь иллюстрация: подобного рода задачи решаются несколькими иными способами.

Листинг 1.17. Решение простого уравнения

```
#include<iostream>
using namespace std;
int main(){
    //Параметры уравнения:
    double a,b;
    //Логическая переменная для записи проверяемых условий:
    bool state;
    //Ввод параметров уравнения:
    cout<<"a = ";
    cin>>a;
    cout<<"b = ";
    cin>>b;
    cout<<"x is: ";
    //Проверка условий и поиск решения:
    state=(a!=0);
    state?(cout<<b/a<<endl,exit(0)):state=(a==0)&&(b==0);
    cout<<(state?"any number!":"no result!")<<endl;
    return 0;
}
```

В программе используются переменные *a* и *b* для записи параметров уравнения, а также логическая переменная *state* для записи результатов проверки условий, которые определяют решение уравнения. После ввода параметров уравнения командой `state=(a!=0)` переменной *state* присваивается результат проверки условия $a \neq 0$. Если условие верно, значение переменной *state* равно `true`, а если условие не верно, то значение переменной *state* равно `false`. При верном условии $a \neq 0$ в результате выполнения команды `state?(cout<<b/a<<endl,exit(0)):state=(a==0)&&(b==0)` отображается значение корня уравнения b/a и завершается работа программы (инструкция `exit(0)`, нулевой аргумент функции `exit()` означает, что программа завершила работу корректно). Поскольку первый блок тернарного оператора состоит из нескольких выражений, они разделяются запятой и заключаются в круглые скобки. Если условие $a \neq 0$ не выполняется (т.е. коэффициент *a* нулевой), значение переменной *state* изменяется с помощью команды `state=(a==0)&&(b==0)`. На самом деле в данном случае хватило бы и команды `state=(b==0)`, поскольку условие равенства нулю коэффициента *a* выполняется автоматически. Такая «неидеальная» форма оставлена скорее для наглядности.

Далее на экран выводится результат, возвращаемый выражением `state?"any number!":"no result!"`. Проверить функциональность программы читатель может самостоятельно.

■ Атака подводной лодки

Рассмотрим следующую задачу: подводная лодка атакует корабль, который прикрывает корабль охраны. У подводной лодки есть n торпед, которые она выпускает по очереди по кораблю до тех пор, пока торпеда не попадет в корабль или пока не закончатся торпеды. Вероятность попадания торпеды в корабль равна p . После каждого запуска торпеды лодкой корабль охраны производит залп по подводной лодке. Вероятность затопления подводной лодки в результате залпа корабля охранения равна p_1 . Составим программу, в которой будет вычисляться вероятность $P(A)$ события A , состоящего в том, что и корабль, и подводная лодка будут затоплены. При этом используем формулу $P(A) = pp_1 \frac{1 - q^n}{1 - q}$, где введено обозначение $q = (1 - p)(1 - p_1)$.

Программный код приведен в листинге 1.18.

Листинг 1.18. Атака подводной лодки

```
#include<iostream>
#include <cmath>
using namespace std;
int main(){
    //Параметры задачи:
    double p,p1,P,q;
    int n;
    //Ввод параметров задачи:
    cout<<"p = ";
    cin>>p;
    //Проверка корректности значения p:
    (p<0||p>1)?(cout<<"Wrong value!"<<endl,exit(0)):cout<<"p1 = ";
    cin>>p1;
    //Проверка корректности значения p1:
    (p1<0||p1>1)?(cout<<"Wrong value!"<<endl,exit(0)):cout<<"n = ";
    cin>>n;
    //Поиск решения:
    q=(1-p)*(1-p1);
    P=p*p1*(1-pow(q,n))/(1-q);
    cout<<"P = "<<P<<endl;
    return 0;
}
```

Обращаем внимание читателя на наличие в программе проверки корректности вводимых значений для вероятностей. Известно, что вероятность не может быть отрицательной или больше нуля. В тернарном операторе соответствующее условие реализовано в виде $(p < 0 || p > 1)$ и $(p1 < 0 || p1 > 1)$ соответственно. Если значение вероятности выходит за указанные пределы, выводится сообщение об ошибке `Wrong value!` и работа программы завершается. Проверить функциональность программы читатель может самостоятельно.

Резюме

Всегда запоминается последняя фраза разговора.

Из к/ф «Семнадцать мгновений весны»

1. Язык программирования C++ является расширением языка программирования C для поддержки объектно-ориентированной парадигмы. В C++ может реализовываться как процедурный подход, так и объектно-ориентированный.
2. Программа в C++ содержит один и только один метод `main()`, выполнение которого отождествляется с выполнением программы. Стандартная программа содержит блок заголовков программы (подключение внешних файлов), блок с объявлением классов и функций, (базовых и производных), прототипами и объявлениями функций, главный метод программы `main()`.
3. Программа помимо прочего может содержать литералы, константы и переменные. При объявлении переменной для нее указывается тип и имя. Для объявления констант используется ключевое слово `const`. Тип литералов определяется автоматически или путем добавления специальных префиксов. Переменная или константа должна быть объявлена и инициализирована до того, как она используется.
4. В C++ существует семь базовых идентификаторов для обозначения типов данных: `int` (целые числа), `float` (числа с плавающей точкой), `double` (числа с плавающей точкой повышенной точности), `char` (символы), `bool` (логический тип) и `void` (используется в функциях, не возвращающих результата). С помощью специальных модификаторов типа параметры базовых типов могут изменяться. Конкретный диапазон значений для типов данных зависит от используемого компилятора.

5. Все операторы, используемые в C++, можно разбить на четыре группы: арифметические, логические, операторы сравнения и побитовые операторы. Данные операторы приведены в таблицах 1.3 – 1.6.
6. В C++ существует тернарный оператор, который во многих отношениях является упрощенной формой условного оператора. Тернарный оператор имеет три операнда и возвращает результат.
7. В качестве оператора присваивания в C++ используется знак равенства (т.е. =). Оператор присваивания возвращает результат – это значение выражения, указанного справа от оператора присваивания.
8. При вычислении выражений в C++ используется автоматическое приведение типов. Приведение типов также может выполняться в явном виде.

Контрольные вопросы

Все те вопросы, которые были поставлены, мы их все соберем в одно место.

В.С. Черномырдин

1. В чем особенность языка C++?
2. Чем принципиально отличается процедурное и объектно-ориентированное программирование?
3. Из каких блоков состоит программа, написанная в C++?
4. Что такое «главный метод программы», как он называется и зачем он нужен?
5. Какие основные типы данных используются в C++?
6. Что такое «модификатор типа»?
7. Как в программе объявляются и инициализируются переменные?
8. Что такое «динамическая инициализация переменной»?
9. Чем константа отличается от переменной? Как константа объявляется?
10. Каким образом в программе определяется тип литерала?
11. Какого типа операторы используются в C++? В чем особенность каждого из типов?

12. Какие арифметические операторы используются в C++?
13. Что такое сокращенная форма арифметического оператора?
14. Что такое оператор инкремента и декремента? Какие существуют формы этих операторов?
15. Какие логические операторы существуют в C++?
16. Какие операторы сравнения используются в C++?
17. Какие основные операторы для выполнения побитовых операций используются в C++?
18. Что такое автоматическое приведение типов? Как и когда оно выполняется?
19. Как в C++ выполняется явное приведение типов?
20. В чем особенности оператора присваивания, используемого в C++?
21. Что такое тернарный оператор? В чем его особенности?

Задачи для самостоятельного решения

Задача 1. Напишите программу, в которой по известной начальной скорости V и времени полета тела T определяется угол α , под которым тело брошено по отношению к горизонту (воспользуйтесь соотношением

$$\alpha = \arcsin\left(\frac{gT}{2V}\right).$$

Задача 2. Для тела, брошенного под углом α к горизонту с начальной скоростью V , определите дальность полета тела L (воспользуйтесь соотношением $L = \frac{V^2 \sin(2\alpha)}{g}$).

Задача 3. Написать программу, в которой по максимальной высоте подъема H и дальности полета L определяется начальная скорость тела V и угол α , под которым тело брошено к горизонту. Воспользоваться соотношениями $tg(\alpha) = 4H/L$ и $V = \sqrt{gL/\sin(2\alpha)}$.

Задача 4. Человек, находящийся на краю обрыва высотой H , бросает с начальной скоростью V камень под углом α к горизонту. Написать программу, которой по введенному пользователем времени t определяется положение камня (высота от дна обрыва $x(t)$ и расстояние до края обрыва $y(t)$). Предусмотреть случай, когда камень упал на дно обрыва (использовать тернарный оператор). Уравнения движения камня имеют вид: вдоль горизонтальной оси $x(t) = Vt \cos(\alpha)$, вдоль направленной вверх вертикальной оси $y(t) = H + Vt \sin(\alpha) - gt^2/2$ (координата отсчитывается от дна обрыва). Время полета камня T определяется условием $y(T) = 0$, т.е. $H + VT \sin(\alpha) - gT^2/2 = 0$, откуда получаем

$$T = \frac{V \sin(\alpha)}{g} \left(1 + \sqrt{1 + \frac{2gH}{V^2 \sin^2(\alpha)}} \right).$$

Задача 5. Самолет летит на высоте H над землей со скоростью U . При полете к наземному объекту он сбрасывает бомбу. Написать программу, которой вычисляется подлетное расстояние S до объекта, на котором производится сбрасывание бомбы. Воспользоваться тем, что бомба до земли летит время $T = \sqrt{2H/g}$. Расстояние S вычисляется как $S = UT$.

Задача 6. Лодка плывет из пункта А в пункт Б и обратно. Скорость лодки V , расстояние между пунктами S и скорость течения реки U задаются пользователем. Написать программу, в которой вычисляется общее время движения лодки T . Учесть, что при движении по течению лодка плывет время $t_1 = S/(V + U)$, а при движении против течения на путь между пунктами уходит время $t_2 = S/(V - U)$. Общее время $T = t_1 + t_2$.

Задача 7. Мотоциклист едет из пункта А в пункт Б и обратно. Написать программу, в которой по известной средней скорости движения V , расстоянию S между пунктами и времени движения t из пункта А в пункт Б вычислить среднюю скорость V_1 мотоциклиста при движении из пункта А в пункт Б и скорость V_2 при движении из пункта Б в пункт А (воспользуйтесь соотношениями $V_1 = S/t$ и $V_2 = S/(2S/V - t)$).

Задача 8. Поезд метро едет на протяжении времени t с постоянной скоростью V , а затем такой же промежуток времени тормозит до полной остановки. Написать программу для вычисления пути S , пройденного поездом. При составлении программы воспользуйтесь набором соотношений: $S = S_1 + S_2$ (S_1 и S_2 пути, пройденные поездом при равномерном и равнозамедленном движении соответственно), $S_1 = Vt$, $S_2 = at^2/2$, где $a = V/t$ – ускорение при равнозамедленном движении. Для сравнения рассчитайте результат по соотношению $S = 3Vt/2$.

Задача 9. Математический маятник совершает колебания по закону $x(t) = A \sin(\omega t + \varphi_0)$. Частота ω маятника известна. В начальный момент координата x в k раз меньше амплитуды A . В какой момент времени T отклонение маятника максимально? Написать программу для определения параметра T при условии, что параметры ω и k вводятся пользователем. При составлении программы воспользоваться соотношениями $\varphi_0 = \arcsin(1/k)$ и $T = (\pi/2 - \varphi_0)/\omega$.

Задача 10. Написать программу для вычисления ускорения свободного падения g как функции h высоты над поверхностью Земли на основании значений гравитационной постоянной $G \approx 6.672 \times 10^{-11}$ (Нм²/кг²), массы Земли $M \approx 5.96 \times 10^{24}$ (кг) и ее радиуса $R \approx 6.37 \times 10^6$ (м). Воспользоваться тем, что на тело массой m действует сила $F = mg$, которая по закону всемирного тяготения равна $F = G \frac{mM}{(R+h)^2}$, откуда получаем

$$g = G \frac{M}{(R+h)^2}.$$

Задача 11. Написать программу для вычисления корня квадратного из комплексного числа $z = x + iy$. Такая операция на множестве комплексных чисел, как известно, имеет два решения: $z_1 = \sqrt{|z|} \exp(i\varphi/2)$ и $z_2 = \sqrt{|z|} \exp(i\varphi/2 + i\pi)$, где $|z| = \sqrt{x^2 + y^2}$ есть модуль комплексного числа z , а φ – его аргумент. Вычислить действительную и мнимую части чисел z_1 и z_2 . Параметры x и y вводятся пользователем с клавиатуры.

Задача 12. Написать программу для расчета ускорения тела массой m , которое находится на горизонтальной плоскости и к которому под углом α (к горизонту) приложена сила F_0 . Коэффициент трения тела о плоскость равен μ . Предусмотреть вариант, когда тело неподвижно (использовать тернарный оператор). Воспользоваться тем, что если тело движется, на него вдоль плоскости движения действует равнодействующая сила $F = F_0 \cos(\alpha) - F_t$, где сила трения $F_t = \mu mg \sin(\alpha)$. Ускорение тела a ищется из второго закона Ньютона $F = ma$.

Задача 13. Тело соскальзывает с наклонной плоскости (угол наклона α) с ускорением a . Написать программу для определения коэффициента трения μ тела о плоскость. Воспользоваться тем, что по второму закону Ньютона $ma = mg \sin(\alpha) - F_t$, где сила трения $F_t = \mu mg \cos(\alpha)$. Отсюда получаем $a = g \sin(\alpha) - \mu \cos(\alpha)$.

Задача 14. Определить силу трения F_t , которая действует на тело массой m , находящееся на наклонной плоскости (угол наклона α). Воспользоваться соотношением $F_t = \mu mg \cos(\alpha)$ для случая, если тело скользит по плоскости, и $F_t = mg \sin(\alpha)$ если тело находится в покое. Использовать тернарный оператор.

Задача 15. Тело массой m находится на наклонной плоскости (угол наклона α) и привязано с помощью нерастяжимой нити (нить направлена вдоль наклонной плоскости). Коэффициент трения тела о плоскость равен μ . Написать программу, для вычисления силы натяжения нити F . Принять во внимание, что возможны две ситуации. Если $tg(\alpha) < \mu$, натяжение нити равно нулю, т.е. $F = 0$. В противном случае сила натяжения нити $F = mg(\sin(\alpha) - \mu \cos(\alpha))$. Использовать тернарный оператор.

Задача 16. В баллоне под поршнем с идеальным газом находится сыпучее вещество. Написать программу для вычисления объема сыпучего вещества V , если известно, что при объеме под поршнем V_1 давление газа равно P_1 , а при объеме под поршнем V_2 давление газа равно P_2 . Использовать соотношение $P_1(V_1 - V) = P_2(V_2 - V)$. С помощью тернарного оператора обработать ситуацию, когда числовые значения V_1 , V_2 , P_1 и P_2 являются некорректными (расчетное значение $V < 0$).

Задача 17. Написать программу для вычисления работы выхода электрона A , если электрон выбивается фотоном с частотой ν , скорость электрона на выходе V . Воспользоваться соотношением $h\nu = A + m_e V^2/2$, где масса электрона $m_e \approx 9.1 \times 10^{-31}$ (кг), а постоянная Планка $h \approx 6.626 \times 10^{-34}$ (Дж·с). Предусмотреть ситуацию, когда введены такие значения ν и V , что $h\nu < m_e V^2/2$ (использовать тернарный оператор).

Задача 18. Написать программу для вычисления кинетической энергии E_k релятивистского электрона. Использовать формулу

$$E_k = \frac{m_e c^2}{\sqrt{1 - (v/c)^2}} - m_e c^2,$$

где масса (масса покоя) электрона $m_e \approx 9.1 \times 10^{-31}$ (кг), скорость света $c \approx 2.998 \times 10^8$ (м/с), а v – скорость электрона. Для сравнения рассчитайте нерелятивистское выражение $E = \frac{m_e v^2}{2}$.

Задача 19. Написать программу для вычисления числа распавшихся ядер ΔN за время t , если в начальный момент ядер было N , а период полураспада равен T . Использовать формулу $\Delta N = N(1 - \exp(-t \ln(2)/T))$. Предусмотреть в программе возможность вычисления значения ΔN как в абсолютных единицах, так и в процентном отношении к начальному количеству ядер N .

Задача 20. Группа из n подводных лодок атакует авианосец. Лодки по очереди выпускают по одной торпедой с ядерным зарядом: если выпущенная лодкой торпеда не попала в авианосец, торпеду выпускает следующая подводная лодка. Вероятность попадания торпеды в авианосец равна p . Выпуская торпеду, лодка демаскирует себя, в силу чего подвергается атаке со стороны кораблей охранения. Вероятность затопления демаскированной подводной лодки кораблями охранения равна p_1 . Написать программу для вычисления вероятности $P(A)$ события A , состоящего в том, что авианосец будет затоплен, а подводные лодки уйдут из зоны действия кораблей охранения без потерь. Использовать формулу $P(A) = p(1 - p_1) \frac{1 - q^n}{1 - q}$, где введено обозначение $q = (1 - p)(1 - p_1)$.

Глава 2

Управляющие инструкции

Самое трудное искусство – это искусство управлять.

К. Вебер

Под управляющими инструкциями обычно подразумевают операторные конструкции, с помощью которых в программе реализуются точки ветвления. Выделяют несколько групп управляющих инструкций, среди которых наиболее важными и часто используемыми являются, пожалуй, условные операторы и операторы цикла.

Условный оператор `if ()`

Если вы можете это вообразить, вы можете это и сделать.

Уолт Дисней

В C++ два условных оператора: `if ()` и `switch ()`. **Оператор `if ()` позволяет выполнять разные блоки операторов в зависимости от того, выполняется ли некое условие.** Условие указывается в круглых скобках после ключевого слова `if`. Общий синтаксис вызова оператора следующий:

```
if(условие) {операторы 1}  
else {операторы 2}
```

Если условие, указанное после ключевого слова `if`, верно, выполняется блок операторов операторы 1. В противном случае выполняется блок операторов операторы 2, указанных после ключевого слова `else`. После выполнения условного оператора управление передается оператору, следующему после него. На рис. 2.1 показана структурная схема, иллюстрирующая работу условного оператора.

Допускается использование упрощенного варианта условного оператора, в котором отсутствует ветка `else` для выполнения операторов при невыполнении условия. Синтаксис вызова условного оператора в такой форме имеет вид

```
if(условие) {операторы 1}
```

В этом случае при выполнении условия управление передается блоку операторов, указанному после ключевого слова `if`. Если условие не выполнено,

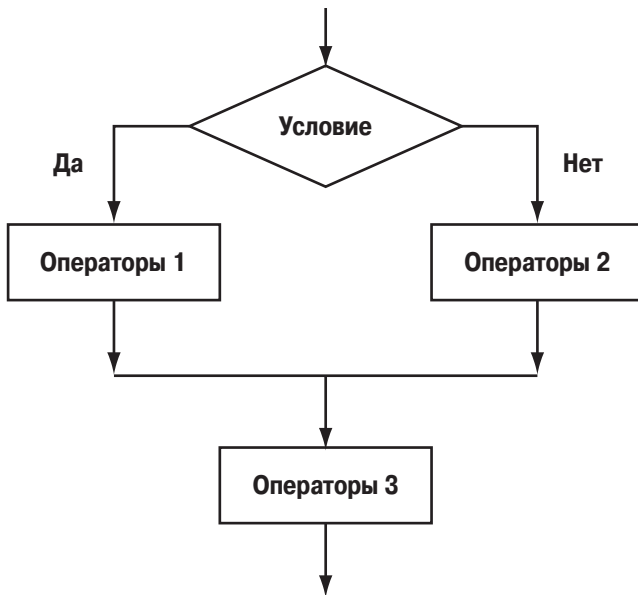


Рис. 2.1. Работа условного оператора

выполняются операторы, размещенные после условного. На рис. 2.2 показана структурная схема работы упрощенного варианта условного оператора.

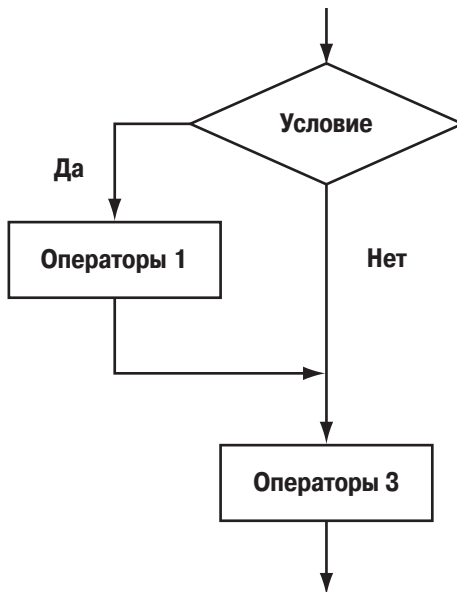


Рис. 2.2. Схема работы упрощенного варианта условного оператора

Нередко на практике используют комбинацию из нескольких вложенных условных операторов. На рис. 2.3 представлена структурная схема работы блока из условных операторов.

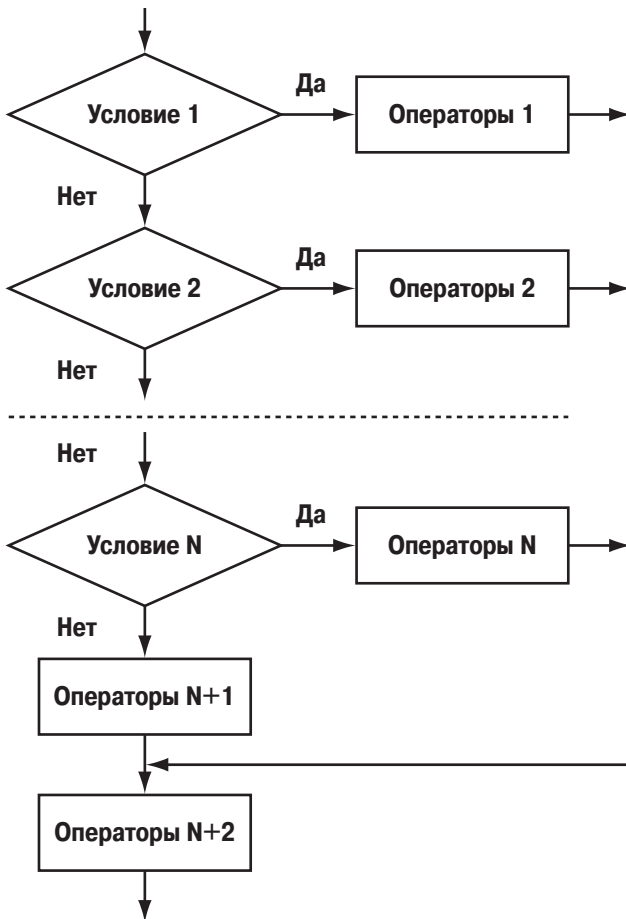


Рис. 2.3. Блок-схема использования нескольких вложенных условных операторов

Синтаксис вызова вложенных условных операторов таков:

```

if(условие 1) {операторы 1}
  else if(условие 2) {операторы 2}
    else if(условие 3) {операторы 3}
      .....
        else if(условие N) {операторы N}
          else {операторы N+1}
  
```

В листинге 2.1 приведен пример простой программы с условным оператором.

Листинг 2.1. Программа с условным оператором

```
#include <iostream>
using namespace std;
int main(){
    double x,y;
    cout<<"x=";
    cin>>x;
    cout<<"y=";
    cin>>y;
    if(y!=0) cout<<"x/y ="<<x/y<<"\n";
    else cout<<"Division by zero!\n";
    return 0;
}
```

В программе объявляются две переменные x и y типа `double`. Значения этих переменных определяются пользователем (считываются с клавиатуры). Если значение переменной y отлично от нуля, вычисляется отношение переменных x/y и соответствующее значение выводится на экран. В противном случае на экране появляется текстовое сообщение `Division by zero!`. Для реализации такого алгоритма в программе использован условный оператор. Проверяемым условием является отличие переменной y от нуля (инструкция `if (y!=0)`). Если это так, выполняется команда `cout<<"x/y ="<<x/y<<"\n"` (на экран выводится отношение двух действительных величин). Если условие не выполнено (то есть если переменная y равна нулю), на экране печатается сообщение о делении на ноль (инструкция `cout<<" Division by zero!\n"` после ключевого слова `else`).

Обращаем внимание читателя на то обстоятельство, что вместо условия `y!=0` с таким же успехом можно указать переменную y , то есть вполне приемлема следующая форма реализации условного оператора:

```
if(y) cout<<"x/y ="<<x/y<<"\n";
else cout<<" Division by zero!\n";
```

Напомним, что в C++ любое ненулевое значение интерпретируется как `true`, а нулевое – как `false`. В этом кроется причина такого демократизма в передаче условий для проверки в условных операторах. Отметим, что данный механизм во многих случаях позволяет существенно упрощать программные коды.

Еще одна программа с условным оператором приведена в листинге 2.2.

Листинг 2.2. Угадывание числа

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
int main(){
    int n,m;
    n=rand()%100+1;
    cout<<"Enter number m=";
    cin>>m;
    cout<<"n="<<n<<" : ";
    if(m>n) cout<<"your number is greater!\n";
    else if(n>m) cout<<"your number is less!\n";
    else cout<<"you have guessed right!\n";
    return 0;
}
```

В программе генерируется случайное число в диапазоне от 1 до 100 включительно (командой `n=rand()%100+1` возвращается остаток от деления случайного числа на 100 плюс единица). При этом использована встроенная функция `rand()` генерирования целых случайных чисел. Чтобы эта функция стала доступной, необходимо в заголовке программы добавить инструкцию `#include <cstdlib>` для подключения библиотеки `cstdlib`. Далее пользователь вводит с клавиатуры целое число, которое сравнивается со сгенерированным в программе. В данном случае код реализован с помощью двух вложенных условных операторов. В зависимости от соотношения чисел выводятся различные сообщения. В начале сообщения выводится значение случайного числа. Далее, если введенное пользователем число больше случайного, выводится сообщение `your number is greater!` (*ваше число больше*). Если введенное пользователем число меньше случайного, выводится сообщение `your number is less!` (*ваше число меньше*). Наконец, если числа совпадают, на экране появится сообщение `you have guessed right!` (*вы угадали число*).

Условный оператор `switch()`

Как сказал один судья, мелких преступников садить в тюрьму, крупным рубить головы, а самых опасных – женить, чтобы по-дольше мучились.

Из к/ф «Сильва»

В тех случаях, когда проверяется больше одного условия, вместо нескольких вложенных условных операторов `if()` нередко используют оператор `switch()`. Синтаксис вызова оператора `switch()` следующий:

```
switch(выражение){
    case значение1:
        операторы
```

```

    break;
case значение2:
    операторы
    break;
...
default:
    операторы
}

```

В круглых скобках после ключевого слова `switch` указывается выражение, значение которого проверяется. Результатом выражения может быть целое число или символ. Значение, возвращаемое выражением, сравнивается со значениями, указанными после ключевых слов `case`. Если имеет место совпадение, выполняется соответствующий блок операторов. Операторы выполняются до конца оператора `switch()` или пока не встретится инструкция `break` (в общем случае инструкция `break` используется для выхода из оператора цикла и перехода к следующему оператору). Если совпадения нет, выполняются операторы после инструкции `default`.

На рис. 2.4 представлена схема работы оператора выбора `switch()`.

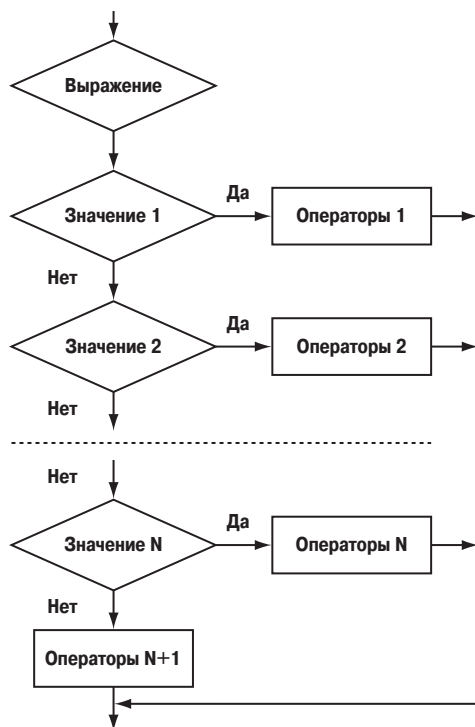


Рис. 2.4. Схема работы оператора выбора `switch()`

Представленная схема приведена в предположении, что в каждом case-блоке использована инструкция `break`, а в конце `switch`-оператора использована инструкция `default`. Отметим, что эта инструкция не является обязательной, также как и инструкции `break`. В листинге 2.3 приведен пример программного кода с использованием `switch`-оператора.

Листинг 2.3. Программа с оператором `switch` ()

```
#include <iostream>
using namespace std;
int main() {
    int n;
    cout<<"Enter n=";
    cin>>n;
    switch(n) {
        case 1:
            cout<<"First case-block\n";
            break;
        case 2:
            cout<<"Second case-block\n";
            break;
        case 3:
            cout<<"Third case-block\n";
            break;
        default:
            cout<<"By default\n";
    }
    return 0;
}
```

В программе с клавиатуры вводится значение для целочисленной переменной `n`. Далее в `switch`-операторе проверяется значение этой переменной. Выделяются три значения этой переменной (1, 2 и 3 соответственно), а также предусмотрен `default`-блок для обработки ситуации, не предусмотренной в `case`-блоках. Результат выполнения программы прост: в зависимости от введенного числа выводится сообщение соответствующего содержания. Если пользователь вводит целое число от 1 до 3 включительно, выводится сообщение `First case-block`, `Second case-block` и `Third case-block` соответственно. В противном случае (т.е. когда введенное пользователем число не равно 1, 2 или 3) выводится сообщение `By default`.

Особенность механизма выхода из оператора `switch` (имеется в виду выход из оператора с помощью инструкции `break`) позволяет объединять несколько `case`-условий. Пример приведен в листинге 2.4.

Листинг 2.4. Числа Фибоначчи

```

#include <iostream>
using namespace std;
int main() {
    int n;
    cout<<"Enter number from 0 to 10: ";
    cin>>n;
    switch(n) {
        case 0:
            cout<<"The number is zero!\n";
            break;
        case 1:
        case 2:
        case 3:
        case 5:
        case 8:
            cout<<"This is Fibonacci number!\n";
            break;
        default:
            cout<<"This is integer number!\n";
    }
    return 0;
}

```

В процессе выполнения программы пользователю предлагается ввести целое число в диапазоне от 0 до 10. Число считывается с клавиатуры и выполняется проверка на предмет того, является ли оно нулем и принадлежит ли последовательности чисел Фибоначчи. Напомним, что последовательность Фибоначчи получается так: первые два числа последовательности равны 1, а каждое последующее равно сумме двух предыдущих. Начальные числа в последовательности Фибоначчи, таким образом, равны 1, 1, 2, 3, 5 и 8 (это те числа, что попадают в диапазон от 0 до 10).

После ввода числа с клавиатуры оно записывается в целочисленную переменную `n`. Значение переменной проверяется в операторе `switch()`. Первый `case`-блок оператора соответствует ситуации, когда пользователем введено нулевое значение. Если значение переменной `n` равно 0, на экран выводится сообщение `The number is zero!`. Благодаря инструкции `break` после оператора вывода указанного сообщения, работа `switch`-оператора на этом завершится. Если же было введено ненулевое значение, первый `case`-блок не выполняется. Вместо этого будет продолжена проверка значения переменной `n` на предмет совпадения с одним из значений 1, 2, 3, 5 или 8. Обращаем внимание читателя на то обстоятельство, что `case`-блоки для значений 1, 2, 3 и 5 являются пустыми, т.е. после инструкции `case` с указанием соответ-

ствующего значения сразу следует следующая `case`-инструкция. Поскольку в `switch`-операторе команды выполняются до первой инструкции `break` (или до окончания всего оператора), такой синтаксис приводит к тому, что для значений 1, 2, 3, 5 и 8 переменной `n` выполняется одна и та же последовательность команд – это те команды, что указаны в `case`-блоке для значения переменной `n` равного 8: появится сообщение `This is Fibonacci number!` и работа `switch`-оператора будет завершена (из-за инструкции `break` после команды `cout<<"This is Fibonacci number!\n"`). Например, если пользователь ввел число 1, в соответствии с приведенным в листинге 2.4 программным кодом должны выполняться все команды после инструкции `case 1:` в `switch`-операторе. Команды выполняются до первой инструкции `break` (или, если не встретится ни одна инструкция `break`, до конца всего `switch`-оператора). Поскольку первая инструкция `break` встречается в блоке `case 8:`, будут выполнены все команды, находящиеся между инструкцией `case 1:` и этой инструкцией `break`.

Наконец, если пользователь ввел ненулевое число, не совпадающее с числами 1, 2, 3, 5 и 8, выполняется блок команд, размещенных после инструкции `default`. В данном случае там всего одна команда, которой на экран выводится сообщение `This is integer number!`.

Оператор цикла `for` ()

Необычные случаи обычно повторяются.

Карел Чапек

Операторы цикла позволяют многократно выполнять серии однотипных действий. Действия выполняются до тех пор, пока остается справедливым (или пока не будет выполнено) некоторое условие. Знакомство с операторами цикла начнем с оператора `for` (). Общий синтаксис вызова оператора `for` () следующий:

`for`(инициализация; условие; изменение переменных) {команды}

В круглых скобках после ключевого слова `for` указывается программный код из трех блоков (при этом каждый из блоков может быть пустым). Блоки разделяются точкой с запятой. Первый блок является блоком инициализации. В нем обычно присваиваются начальные значения для переменной (или переменных) цикла. Второй блок – условие выполнения оператора цикла. Пока справедливо условие, оператор цикла будет выполняться. Третий блок – это блок изменения индексных переменных. Указанное на значение блоков достаточно условное. Детально назначение и возможное

использование различного синтаксиса оператора цикла `for()` рассмотрим на примерах. Здесь отметим общий принцип выполнения оператора цикла: сначала выполняются команды, указанные в первом блоке оператора `for()`. После этого проверяется условие, указанное во втором блоке оператора. Если условие справедливо, выполняются команды после инструкции `for()` (если команд несколько, они заключаются в фигурные скобки). После выполнения команд в фигурных скобках выполняются команды третьего блока в круглых скобках после ключевого слова `for`. Далее снова проверяется условие (второй блок). При справедливости условия снова выполняются команды в фигурных скобках и команды третьего блока и т.д. Схема выполнения оператора цикла проиллюстрирована на рис. 2.5.

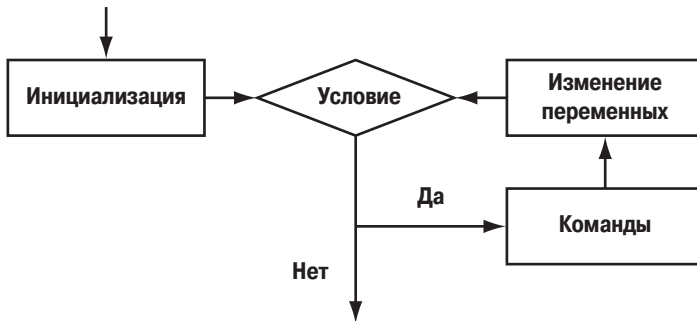


Рис. 2.5. Схема работы оператора цикла `for()`

Один из наиболее простых способов использования оператора цикла для расчета суммы натуральных чисел приведен в листинге 2.5.

Листинг 2.5. Вычисление суммы натуральных чисел

```

#include <iostream>
using namespace std;
int main(){
    int n,i,s=0;
    cout<<"Enter number n=";
    cin>>n;
    for(i=1;i<=n;i++){
        s+=i;
    }
    cout<<"Sum of natural numbers is: "<<s<<"\n";
    return 0;
}
  
```

Основу программы составляет оператор цикла `for(i=1;i<=n;i++){s+=i;}`, который содержит в первом блоке команду инициализации

индексной переменной $i=1$ с начальным единичным значением. Вторым блоком – проверяемое условие $i \leq n$. Это означает, что оператор цикла выполняется до тех пор, пока индексная переменная i не превышает значения переменной n (значение переменной предварительно вводится с клавиатуры). В третьем блоке указана инструкция $i++$, в силу чего значение индексной переменной увеличивается на единицу. Наконец, в основном блоке оператора цикла (в фигурных скобках) использована команда $s+=i$, которой на каждом шаге целочисленная переменная s (начальное нулевое значение этой переменной установлено при ее объявлении) увеличивается на значение индексной переменной i . Сразу отметим, что поскольку в основном блоке оператора цикла команда всего одна, фигурные скобки можно было не использовать.

Процедура выполнения оператора цикла следующая. Сначала индексной переменной присваивается единичное значение. Затем проверяется условие, и если индексная переменная меньше значения n , выполняется команда $s+=i$ (с текущими значениями переменных s и i). Далее выполняется команда $i++$, что приводит к увеличению на единицу индексной переменной, после чего снова проверяется условие. Процесс будет продолжаться до тех пор, пока значение индексной переменной не превысит значения переменной n . Таким образом, после выполнения оператора цикла значение переменной s определяется суммой натуральных чисел от 1 до n . Это значение выводится на экран.

Как отмечалось выше, некоторые (а то и все) блоки в круглых скобках после ключевого слова `if` могут быть пустыми. Например, при составлении программы допускается использовать и несколько иной синтаксис вызова оператора цикла. Соответствующий программный код приведен в листинге 2.6.

Листинг 2.6. Вычисление суммы натуральных чисел

```
#include <iostream>
using namespace std;
int main() {
    int n, i=1, s=0;
    cout<<"Enter number n=";
    cin>>n;
    for(; i<=n; i++) {
        s+=i;
    }
    cout<<"Sum of natural numbers is: "<<s<<"\n";
    return 0;
}
```

По сравнению с предыдущим случаем, индексная переменная i при объявлении получает начальное значение 1, а в операторе цикла отсутствует первый блок (блок инициализации). В частности, оператор цикла имеет вид `for (; i<=n; i++) { s+=i; }`. В этом случае необходимости в первом блоке инициализации значений нет, поскольку такая инициализация для индексной переменной выполнена при ее объявлении. Еще один пример реализации оператора цикла для вычисления суммы натуральных чисел представлен в листинге 2.7.

Листинг 2.7. Вычисление суммы натуральных чисел

```
#include <iostream>
using namespace std;
int main() {
    int n, i=1, s=0;
    cout<<"Enter number n=";
    cin>>n;
    for(; i<=n; ) {
        s+=i;
        i++;
    }
    cout<<"Sum of natural numbers is: "<<s<<"\n";
    return 0;
}
```

В инструкции `for (; i<=n;)` отсутствуют как первый, так и третий блоки. Команда изменения на единицу значения индексной переменной вынесена в основной блок оператора цикла. В принципе, можно отказаться и от блока с условием выполнения цикла. Соответствующий программный код представлен в листинге 2.8.

Листинг 2.8. Вычисление суммы натуральных чисел

```
#include <iostream>
using namespace std;
int main() {
    int n, i=1, s=0;
    cout<<"Enter number n=";
    cin>>n;
    for(;;) {
        s+=i;
        i++;
        if(i>n) break;
    }
    cout<<"Sum of natural numbers is: "<<s<<"\n";
    return 0;
}
```

Хотя инструкция `for (; ;)` выглядит довольно странно, тем не менее функциональность программного кода, представленного в листинге 2.8, такая же, как и в предыдущих случаях. Однако теперь, поскольку все три блока в круглых скобках после инструкции `for` отсутствуют, инициализацию, изменение значения индексной переменной и проверку условия необходимо реализовывать при объявлении переменных, до начала оператора цикла и в основном блоке оператора цикла. Причем пользователь должен самостоятельно предусмотреть возможность завершения оператора цикла! В данном случае цикл будет завершен благодаря инструкции `if (i>n) break` в основном блоке оператора `for ()`. Как только значение переменной `i` превысит значение `n`, будет выполнен оператор `break`, в результате чего выполнение оператора цикла будет завершено.

Если блок в `for`-операторе содержит несколько команд, они разделяются запятой. Более того, в операторе цикла `for ()` в блоке инициализации можно объявлять переменные. Используя эти свойства оператора цикла `for ()`, часто добиваются значительного сокращения программного кода. В листинге 2.9 приведен еще один пример программы с вычислением суммы натуральных чисел.

Листинг 2.9. Вычисление суммы натуральных чисел

```
#include <iostream>
using namespace std;
int main() {
    int n;
    cout<<"Enter number n=";
    cin>>n;
    for(int i=1, s=0; i<=n; s+=i++);
    cout<<"Sum of natural numbers is: "<<s<<"\n";
    return 0;
}
```

Если не принимать во внимание ту часть программы, что связана с вводом пользователем значения переменной `n` и выводом значения суммы `s` на экран, основная часть программы состоит из инструкции `for(int i=1, s=0; i<=n; s+=i++)`. В блоке инициализации объявляются с начальными значениями две переменные: `i` со значением 1 и `s` со значением 0. Проверяемое условие `i<=n` является традиционным. Третий блок состоит из одной команды `s+=i++`, которой значение индексной переменной `i` увеличивается на единицу, а значение переменной `s` увеличивается на величину индексной переменной. В команде `s+=i++` оператор инкремента использован в постфиксной форме. Поэтому сначала при текущем значении переменной `i` вычисляется выражение `s+=i`, после чего переменная `i` увеличивается на единицу. Если заменить постфиксную форму оператора инкремента на пре-

фиксную (т.е. использовать команду `s+=++i`), алгоритм изменится. Сначала на единицу будет изменяться значение индексной переменной, а уже после этого, на основе измененного значения, будет рассчитываться значение переменной `s`. В результате будет вычислена сумма натуральных чисел не от 1 до `n`, а от 2 до `n+1`. Желаящие могут убедиться в этом самостоятельно.

Индексная переменная в операторе цикла – понятие достаточно условное. Обычно под такой переменной подразумевают целочисленную переменную, которая пробегает набор дискретных значений. Однако это не всегда так. В листинге 2.10 приведен пример программы, в которой роль индексной переменной, если этот термин вообще здесь уместен, выполняет переменная типа `char`.

Листинг 2.10. Угадывание буквы

```
#include <iostream>
using namespace std;
int main() {
    for(char x='a'; x!='z';) {
        cout<<"Guess a symbol: ";
        cin>>x;
    }
    cout<<"Correct!\n";
    return 0;
}
```

Основу программы составляет оператор цикла с заголовком `for(char x='a'; x!='z';)`, в котором переменная `x` типа `char` инициализируется со значением `'a'`. Проверяемым условием является `x!='z'`, что означает продолжение оператора цикла до тех пор, пока значение переменной `x` не станет равным `'z'`. Третьего блока вообще нет. Изменение значения переменной `x` происходит в результате считывания с клавиатуры (команды `cout<<"Guess a symbol: "` и `cin>>x` в основном блоке оператора цикла). Таким образом, на экране будет отображаться фраза `Guess a symbol` (*угадайте букву*), после чего пользователь должен ввести букву. Процесс продолжается до тех пор, пока не будет введена буква `z`. После этого отображается фраза `Correct!` (*правильно*) (предпоследняя команда `cout<<"Correct!\n"` в программе). В принципе, тот же оператор цикла можно записать более компактно: `for(char x='a'; x!='z'; cout<<"Guess a symbol: ", cin>>x)`. Фактически, в таком синтаксисе две команды основного блока оператора цикла перенесены в блок изменения значения переменных. С функциональной точки зрения оба кода эквивалентны.

Хочется обратить внимание читателя на использование вложенных операторов цикла. Речь идет о том, что в рамках одного оператора цикла использует-

ся другой оператор цикла. К такой ситуации приводят самые разные задачи. В качестве иллюстрации рассмотрим программу по распечатке натуральных чисел от 1 до 15 по пяти столбикам. Код представлен в листинге 2.11.

Листинг 2.11. Распечатка натуральных чисел

```
#include <iostream>
using namespace std;
int main(){
    int i, j;
    for(i=1; i<=3; i++){
        for(j=1; j<=5; j++) cout<<3*(j-1)+i<<" ";
        cout<<"\n";
    }
    return 0;
}
```

Результат выполнения программы будет выглядеть как

```
1 4 7 10 13
2 5 8 11 14
3 6 9 12 15
```

Индексная переменная *i* внешнего оператора цикла принимает значения от 1 до 3 включительно. Эта переменная определяет номер строки, в которой отображается число. Номер столбца определяется индексной переменной *j* внутреннего оператора цикла. Эта переменная при каждом фиксированном значении переменной *i* пробегает значения от 1 до 5. Во внутреннем операторе цикла выполняется всего одна команда `cout<<3*(j-1)+i<<" "`, с помощью которой распечатываются числа в соответствующей строке. Что касается внешнего оператора цикла, то в рамках каждого цикла выполняется две команды: внутренний оператор цикла (распечатываются числа в строке) и команда `cout<<"\n"` (для перехода к новой строке).

Следует четко понимать разницу между двумя (или более) вложенными операторами цикла и одним оператором цикла с несколькими индексными переменными. Пример такой ситуации проиллюстрирован в листинге 2.12.

Листинг 2.12. Две индексных переменных

```
#include <iostream>
using namespace std;
int main(){
    int i, j;
    for(i=10, j=90; i<j; i+=5, j-=10)
        cout<<i<<" "<<j<<"\n";
    return 0;
}
```

Результатом выполнения программы будет два столбика чисел:

```
10 90
15 80
20 70
25 60
30 50
35 40
```

Дело в том, что, в отличие от случая вложенных операторов, в данной ситуации обе индексные переменные *i* и *j* изменяются синхронно. Переменная *i* инициализируется со значением 10, а переменная *j* – со значением 90. За каждый цикл значение переменной *i* увеличивается на 5, а значение переменной *j* уменьшается на 10. Значения переменных выводятся на экран. Процесс продолжается до тех пор, пока значение переменной *i* меньше значения переменной *j*.

Оператор цикла `while()`

*Лидусик, пока вы тут заседаете,
у нас машину угнали.*

Из к/ф «Гараж»

Помимо оператора цикла `for()`, широко используются циклы `while()` и `do-while()`. Синтаксис вызова оператора `while()` следующий:

```
while (условие) {
    команды
}
```

Сначала проверяется условие, указанное в круглых скобках после ключевого слова `while`. Если условие справедливо, поочередно выполняются операторы, указанные в фигурных скобках после инструкции `while()`. Если инструкция одна, фигурные скобки можно не указывать. Схема действия оператора цикла `while()` проиллюстрирована на рис. 2.6.

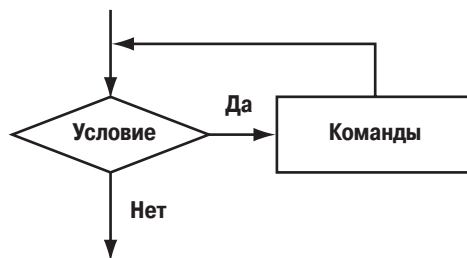


Рис. 2.6. Схема выполнения оператора цикла `while()`

Синтаксис вызова оператора `do-while()` имеет вид

```
do{
    команды
}
while (условие) ;
```

В операторе цикла `do-while()` выполняемые команды (заключенные в фигурные скобки) указываются после ключевого слова `do`. Далее проверяется условие, указанное в круглых скобках после ключевого слова `while`. Если условие выполнено, снова выполняются команды после ключевого слова `do` и т.д. Принцип работы оператора `do-while()` иллюстрирует рис. 2.7.

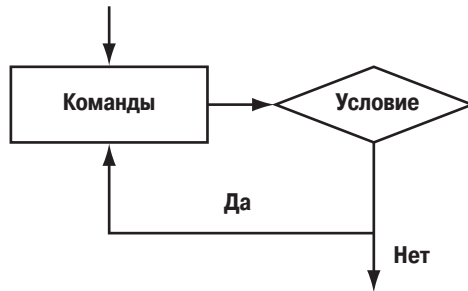


Рис. 2.7. Схема выполнения оператора цикла `do-while()`

Таким образом, принципиальная разница между операторами `while()` и `do-while()` состоит в том, что в первом случае сначала проверяется условие, а затем (если верно условие) выполняются команды. Во втором случае сначала по крайней мере один раз выполняются команды, а затем проверяется условие. По сравнению с оператором цикла `for()`, операторы `while()` и `do-while()` требуют от программиста большей ответственности в первую очередь в плане детальной проработки механизма изменения значения проверяемого условия в процессе выполнения команд основного блока оператора. Программный код должен быть составлен корректно, чтобы не получить бесконечный цикл.

В листинге 2.13 представлена реализация программы для вычисления суммы натуральных чисел с помощью оператора цикла `while()`.

Листинг 2.13. Вычисление суммы натуральных чисел

```
#include <iostream>
using namespace std;
int main(){
    int n,i=1,s=0;
    cout<<"Enter number n=";
    cin>>n;
```

```

while (i<=n) {
    s+=i;
    i++;}
cout<<"Sum of natural numbers is: "<<s<<"\n";
return 0;
}

```

В принципе, вместо двух команд `s+=i` и `i++` можно было использовать одну команду `s+=i++`, но в данном случае это не принципиально. Алгоритм выполнения программы следующий: сначала выводится текстовое сообщение с приглашением ввести число. После ввода пользователем числа проверяется неравенство `i<=n` (переменная `i` предварительно инициализирована с единичным значением). Если значение переменной `i` не превышает введенного пользователем значения `n`, выполняются команды оператора цикла (`s+=i` и `i++` соответственно). После этого снова проверяется условие и т.д. до тех пор, пока значение переменной `i` не превысит значения `n`. Результат вычисления суммы чисел выводится на экран.

В целом процесс вычислений, по сравнению со случаем, когда использовался оператор цикла `for()`, изменился мало. Это же относится и к оператору `do-while()`. Пример использования этого оператора приведен в листинге 2.14.

Листинг 2.14. Вычисление суммы натуральных чисел

```

#include <iostream>
using namespace std;
int main(){
    int n,i=1,s=0;
    cout<<"Enter number n=";
    cin>>n;
    do{
        s+=i;
        i++;
    }while(i<=n);
    cout<<"Sum of natural numbers is: "<<s<<"\n";
    return 0;
}

```

На первый взгляд может показаться, что разницы между двумя программами с циклами `while()` и `do-while()` практически никакой, но одно принципиальное отличие все же есть. Если пользователь введет, например, отрицательное число (значение переменной `n`), то первая программа (листинг 2.13) в качестве значения суммы укажет 0, в то время как во втором случае (листинг 2.14) будет выведена 1. Причина в том, что в цикле `while()` при ложном проверяемом условии команды оператора цикла не

выполняются и в качестве значения суммы возвращается начальное нулевое значение переменной `s`. Во втором случае сначала выполняется один цикл и уже после этого проверяется условие. За этот один выполненный цикл значение переменной `s` увеличивается на 1, и в результате программой для суммы натуральных чисел возвращается единичное значение.

Инструкция безусловного перехода

Уж послала, так послала...

Из м/ф «Падал прошлогодний снег»

В C++ существует инструкция `goto`, которая позволяет выполнять переход к заранее определенному месту программы. Место, к которому осуществляется переход, помечается с помощью специального идентификатора, который называют меткой. Чтобы вставить в программный код метку, необходимо ввести в соответствующем месте имя метки с двоеточием в конце. Чтобы перейти к помеченному месту кода, необходимо после инструкции `goto` указать метку, определяющую место перехода.

Использование инструкций безусловного перехода в программах считается дурным тоном. Существует точка зрения, что наличие в программе инструкций `goto` замедляет процесс выполнения программы и снижает читаемость программного кода. Внимание на этом вопросе заострять не будем. Отметим лишь, что поскольку механизм безусловных переходов в языке программирования существует, необходимо иметь о нем хотя бы общее представление. А использовать или не использовать этот механизм на практике – право читателя.

Работу с инструкциями безусловного перехода проиллюстрируем на примере создания цикла для расчета суммы чисел. В листинге 2.15 приведен код программы, в которой вместо оператора цикла использована конструкция с меткой, условным оператором и инструкцией безусловного перехода.

Листинг 2.15. Вычисление суммы натуральных чисел

```
#include <iostream>
using namespace std;
int main() {
    int n, i=1, s=0;
    cout<<"Enter number n=";
    cin>>n;
    // Используется метка
    mylabel:
```

```

s+=i;
i++;
// Команда с инструкцией безусловного перехода
if(i<=n) goto mylabel;
cout<<"Sum of natural numbers is: "<<s<<"\n";
return 0;
}

```

Начальная часть программы ничем не отличается от рассмотренных ранее вариантов: объявляются целочисленные переменные *i* (начальное значение 1), *s* (начальное значение 0) и *n* (значение вводится пользователем). После метки *mylabel* следует три команды: *s+=i*, *i++* и *if(i<=n) goto mylabel*. Именно с помощью этих команд реализуется цикл. Первые две особых комментариев не требуют. Третьей командой в рамках условного оператора проверяется условие *i<=n*. Если значением выражения является *true*, выполняется безусловный переход к тому месту в программе, что выделено меткой *mylabel*, т.е. к началу рассмотренного блока. Процесс продолжается до тех пор, пока значение переменной *i* не превысит значение переменной *n*. В этом случае в условном операторе команда безусловного перехода не выполняется, а управление передается к следующему после условного оператора. Желаящие могут легко убедиться, что сумма натуральных чисел в результате вычисляется корректно.

Примеры решения задач

Приобретать познания еще недостаточно для человека, надо уметь отдавать их в рост.

И. Гете

Далее рассматривается ряд задач, подразумевающих составление программных кодов, в которых широко используются операторы цикла и условные операторы.

■ Вычисление синуса

Как известно, функция $\sin(x)$ может быть вычислена в виде ряда

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}.$$

На практике при вычислении значения функции $\sin(x)$ соответствующий ряд ограничивают, т.е. рассматривают приближенное выражение

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + \frac{(-1)^N x^{2N+1}}{(2N+1)!} = \sum_{n=0}^N \frac{(-1)^n x^{2n+1}}{(2n+1)!}.$$

Воспользуемся этим выражением при создании программы, в которой вычисляется значение функции $\sin(x)$. Аргумент x вводится пользователем с клавиатуры, а параметр N описывается как целочисленная константа. Программный код приведен в листинге 2.16.

Листинг 2.16. Вычисление синуса

```
#include <iostream>
using namespace std;
//Граница ряда:
const int N=100;
int main(){
    //Аргумент функции и "рабочие" переменные:
    double x,q,s=0;
    //Индексная переменная:
    int n;
    cout<<"Enter x = ";
    cin>>x;
    q=x;
    //Вычисление синуса:
    for(n=1;n<=N;n++){
        s+=q;
        q*=(-1)*x*x/(2*n)/(2*n+1);
    }
    //Результат:
    cout<<"sin(" <<x<<" ) = " <<s<<endl;
    return 0;
}
```

Желающие могут сравнить результат вычисления с помощью описанного алгоритма с результатом вычисления синуса через встроенную функцию.

Обращаем внимание читателя на способ вычисления ряда. Значение суммы ряда записывается в переменную s . В рамках оператора цикла эта переменная изменяется на величину добавки q . Эта добавка фактически является слагаемым ряда, в каждом цикле она меняется. При значении индексной переменной n добавка равняется $q_n = \frac{(-1)^n x^{2n+1}}{(2n+1)!}$. Формально для рас-

чета добавки можно отдельно вычислить x^{2n+1} и $(2n+1)!$, но это плохой подход. Дело в том, что при больших n значение $(2n+1)!$ очень велико, большим может быть и значение выражения x^{2n+1} . Значения быстро становятся настолько большими, что выходят за допустимые пределы. При всем

этом добавка q_n с увеличением n уменьшается: добавка является результатом деления одного большого числа на другое, еще большее число. Поэтому в программе для вычисления добавки используется итерационное соотношение $q_n = q_{n-1} \cdot \frac{(-1) \cdot x^2}{(2n) \cdot (2n+1)}$. Именно это соотношение использовано в программе, благодаря чему удастся избежать проблемы расчета больших числовых значений.

■ Вычисление произведения

Напишем программу для вычисления приближенного значения бесконечного произведения $\prod_{n=2}^{\infty} \left(1 - \frac{2}{n(n+1)}\right) = \frac{1}{3}$. Вычислять будем конечное произведение $\prod_{n=2}^N \left(1 - \frac{2}{n(n+1)}\right)$. Верхняя граница произведения N вводится пользователем. Программный код приведен в листинге 2.17.

Листинг 2.17. Вычисление произведения

```
#include <iostream>
using namespace std;
int main() {
    //Граница произведения:
    int N;
    //Индексная переменная:
    int n;
    //Переменная для записи произведения:
    double s=1;
    //Ввод границы произведения:
    cout<<"Enter N = ";
    cin>>N;
    //Вычисление произведения:
    for (n=2;n<=N;n++)
        s*=(1-(double)2/(n*(n+1)));
    //Результат:
    cout<<"Product is "<<s<<endl;
    return 0;
}
```

Результат выполнения программы может выглядеть, например, следующим образом (жирным шрифтом выделен ввод пользователя):

```
Enter n = 1000
Product is 0.334
```


Что касается непосредственно программного кода, то обращаем внимание читателя на инструкцию явного приведения типа (double) в команде $s*=(1-(\text{double})2/(n*(n+1)))$, выполняемой в рамках оператора цикла. Эта инструкция необходима, поскольку без нее будет выполняться целочисленное деление.

■ Расчет траектории тела

Решим программными методами задачу о движении тела, брошенного под углом к горизонту, однако, в отличие от предыдущей главы, к точным аналитическим результатам прибегать не будем. В частности, составим разностную схему для решения системы дифференциальных уравнений, которыми описывается движение тела. В векторной форме уравнение, описывающее динамику тела массой m , имеет вид $m\ddot{\vec{r}} = m\vec{g}$, где через \vec{r} обозначен радиус-вектор на тело, точка означает производную по времени, а \vec{g} есть вектор ускорения свободного падения. После сокращения на массу в проекциях на координатные оси x (направлена горизонтально) и y (направлена вертикально вверх) получаем систему дифференциальных уравнений $\ddot{x} = 0$ и $\ddot{y} = -g$. Далее рассматриваем дискретные моменты времени $t_n = n\Delta t$ ($n = 0, 1, 2, \dots$, а Δt – интервал дискретности, чем он меньше, тем выше точность вычислений). Будем обозначать x_n и y_n координаты тела в момент времени t_n . Если ввести в рассмотрение два новых параметра v_n (проекция скорости тела на горизонтальную ось в момент времени t_n) и u_n (проекция скорости тела на вертикальную ось в момент времени t_n), то на основе указанной выше системы дифференциальных уравнений можем записать рекуррентные соотношения $v_{n+1} = v_n$, $u_{n+1} = u_n - g\Delta t$, $x_{n+1} = x_n + v_n\Delta t$ и $y_{n+1} = y_n + u_n\Delta t$, с начальными условиями $x_0 = y_0 = 0$, $v_0 = V \cos(\alpha)$, $u_0 = V \sin(\alpha)$ (V – начальная скорость тела, а α – угол, под которым тело брошено к горизонту). Именно этими соотношениями воспользуемся при составлении программы, представленной в листинге 2.18.

Листинг 2.18. Расчет траектории тела

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
    //Ускорение свободного падения и число pi:
    const double g=9.8,pi=3.1415;
    //Шаг дискретности по времени:
    double dt=0.0001;
```

```

//Рабочие переменные:
double V, alpha, t, v, u, x=0, y=0;
int n=0;
//Ввод параметров задачи:
cout<<"Enter V = ";
cin>>V;
cout<<"Enter alpha = ";
cin>>alpha;
//Перевод градусов в радианы:
alpha=alpha*pi/180;
cout<<"Enter t = ";
cin>>t;
//Начальная скорость (проекции):
v=V*cos(alpha);
u=V*sin(alpha);
//Вычисление координат тела:
do{
    n++;
    y+=u*dt;
    x+=v*dt;
    u-=g*dt;
}while((y>0)&&(n*dt<t));
//Вывод результатов с учетом конечности времени полета:
cout<<"y = "<<y<<" : ";
cout<<(t<sqrt(2*V*sin(alpha)/g)?V*sin(alpha)*t-g*t*t/2:0)<<endl;
cout<<"x = "<<x<<" : ";
cout<<(t<sqrt(2*V*sin(alpha)/g)?V*cos(alpha)*t:V*cos(alpha)*t)<<endl;
return 0;
}

```

В программе предусмотрена обработка ситуации, когда введенное пользователем время больше времени полета тела. Критерием падения тела является равенство нулю y – координаты. Координата y равна нулю в начальный момент и в момент удара тела о землю.

Пользователь вводит скорость тела (переменная V), угол (переменная α) и момент времени (переменная t), для которого необходимо рассчитать координаты тела. После этого запускается условный оператор `do-while()`, в котором вычисляются координаты тела: последовательно изменяется время (шаг дискретности по времени определяется переменной dt), определяется скорость (проекция скорости на координатные оси – переменные v и u) и координаты тела в каждый из моментов времени (переменные x и y). Цикл выполняется до тех пор, пока выполнены два условия: $y > 0$ (условие полета тела – пока тело не упало, вертикальная координата больше нуля), и $n*dt < t$ (не достигнут момент времени t). Обращаем внимание читателя, что использование оператора `do-while()`, а не `while()`,

является принципиальным, поскольку в начальный момент координаты тела нулевые, и условие $y > 0$ не выполняется. Если это условие проверить сразу (что делается в операторе `while()`), цикл выполняться не будет. В операторе `do-while()` один цикл выполняется до проверки условия, координаты тела меняются, поэтому такой проблемы не возникает.

Кроме рассчитанного указанным способом результата для сравнения выводится и точное значение координат, рассчитанное по приводившимся ранее формулам (см. главу 1). Вывод аналитического значения осуществляется с учетом того, что пользователь мог ввести слишком большой момент времени, так что тело успевает упасть на землю. Поэтому при выводе значения каждой из координат используется тернарный оператор. Приближенное и точное значение координат разделяются при выводе двоеточием. Результат выполнения программы (для времени, не превышающего время падения тела) может выглядеть, например, так:

```
Enter V = 10
Enter alpha = 60
Enter t = 1
y = 3.76059 : 3.7601
x = 5.00027 : 5.00027
```

Если введен слишком большой интервал времени, получим нечто следующее:

```
Enter V = 10
Enter alpha = 60
Enter t = 100
y = -0.0000283487 : 0
x = 8.83797 : 8.83731
```

Здесь и выше жирным шрифтом выделен ввод пользователя.

■ Решение уравнения методом последовательных итераций

Если алгебраическое уравнение записано в виде $x = \varphi(x)$, то при некоторых обстоятельствах решение уравнения (приближенное) может быть найдено методом последовательных приближений. Суть метода состоит в том, что $n + 1$ -е приближение для корня уравнения x_{n+1} вычисляется на основе n -го приближения x_n через соотношение $x_{n+1} = \varphi(x_n)$. Для этого необходимо, чтобы в некоторой окрестности начального приближения x_0 (этой окрестности принадлежат вычисляемые итерационные значения x_n)

выполнялось соотношение $\left| \frac{d\varphi}{dx} \right| < 1$. Самое сложное, как правило, состоит

в том, чтобы представить уравнение в виде $x = \varphi(x)$. Формально проблем здесь нет – уравнение вида $f(x) = 0$ сводится к указанному виду, если положить $\varphi(x) = x + f(x)$. Однако такой подход не гарантирует выполнение необходимого условия $\left| \frac{d\varphi}{dx} \right| < 1$.

Например, квадратное уравнение $x^2 - 5x + 6 = 0$ имеет два корня $x = 2$ и $x = 3$. Если представить это уравнение в виде $x = \frac{x^2 + 6}{5}$, т.е. функция $\varphi(x) = \frac{x^2 + 6}{5}$, производная которой $\frac{d\varphi(x)}{dx} = \frac{2x}{5}$. Условие $\left| \frac{d\varphi}{dx} \right| < 1$ выполняется для значений $|x| < 2.5$. Поэтому корень $x = 3$ найти не удастся (зато можно найти корень $x = 2$). Чтобы найти корень $x = 3$, исходное уравнение представим в виде $x = \sqrt{5x - 6}$. В этом случае $\varphi(x) = \sqrt{5x - 6}$ и $\frac{d\varphi(x)}{dx} = \frac{5}{2\sqrt{5x - 6}}$. Условие $\left| \frac{d\varphi}{dx} \right| < 1$ выполняется при $x > 2.45$, поэтому корень $x = 3$ может быть найден, в отличие от корня $x = 2$.

В листинге 2.19 приведен программный код, с помощью которого находятся корни указанного уравнения.

Листинг 2.19. Метод последовательных итераций

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    //Корень уравнения:
    double x;
    //Количество итераций и индексная переменная:
    int n=100,i;
    //Вычисление корня x=2:
    x=0;
    for(i=1;i<=n;i++)
        x=(x*x+6)/5;
    cout<<"x = "<<x<<endl;
    //Вычисление корня x=3:
    x=5;
    for(i=1;i<=n;i++)
        x=sqrt(5*x-6);
    cout<<"x = "<<x<<endl;
    return 0;
}
```

В результате выполнения программы получаем

```
x = 2
x = 3
```

Найдены, фактически, точные решения – это результат ошибок округления. Вообще же получаемые таким образом результаты являются приближенными. Чтобы убедиться в этом, достаточно уменьшить значение переменной *n*, определяющей количество итераций.

■ Калькулятор

Рассмотрим программу, которой реализуется очень простой калькулятор. Пользователь поочередно вводит числа и символы операций с этими числами (сложение, вычитание, умножение и деление). Операции выполняются последовательно, до тех пор, пока пользователь не вводит знак равенства. Программный код приведен в листинге 2.20.

Листинг 2.20. Калькулятор

```
#include <iostream>
using namespace std;
int main(){
    //Переменные для записи результата
    //вычислений и вводимого числа:
    double x,s;
    //Переменная для записи символа операции:
    char op;
    //Ввод первого числа:
    cout<<">> ";
    cin>>x;
    s=x;
    //Оператор цикла для вычисления результата:
    while(true){
        //Считывание символа операции:
        cout<<">> ";
        cin>>op;
        //Команда выхода из оператора цикла:
        if(op=='='){
            cout<<"---> "<<s<<endl;
            exit(0);
        }
        //Ввод числа:
        cout<<">> ";
        cin>>x;
```

```

//Обработка символа операции:
switch(op) {
    case '+':
        s+=x;
        break;
    case '-':
        s-=x;
        break;
    case '*':
        s*=x;
        break;
    case '/':
        s/=x; }
}
return 0;
}

```

Что касается реализованного в программе алгоритма, то он достаточно прост. В одну переменную (переменная *s*) записывается результат после каждого действия. В переменную *x* записывается вводимое пользователем число. На основании символа операции (переменная *op*) определяется, какую операцию необходимо выполнить, чтобы с учетом введенного значения *x* изменить результат *s*. Пример выполнения программы может выглядеть так (жирным шрифтом выделен ввод пользователя):

```

>> 3
>> +
>> 2
>> *
>> 4
>> -
>> 1
>> /
>> 5
>> =
---> 3.8

```

Обращаем внимание на формально бесконечный оператор цикла, посредством которого реализуется процесс ввода пользователем. Для выхода из оператора цикла необходимо, чтобы пользователь ввел знак равенства (символы операций записываются в переменную *op* типа *char*). После ввода очередного символа операции выполняется проверка условия, совпадает ли он со знаком равенства. Для сравнения используется условный оператор. Если выполняется условие *op=='='*, результат (переменная *s*) выводится на экран и работа программы завершается.

Выбор нужной арифметической операции осуществляется с помощью оператора `switch()`. Проверяемым выражением является переменная `op`, которая согласно генеральному плану может принимать символьные значения `+`, `-`, `*` и `/`. В зависимости от того, какое значение принимает переменная `op`, выполняется соответствующая операция (сложение, вычитание, умножение и деление).

■ Вычисление объема методом Монте-Карло

В прикладных расчетах методы Монте-Карло применяются достаточно часто. Причем нередко соответствующие методы исследования являются единственно возможными. В данном случае не будем останавливаться на особенностях методов Монте-Карло, ограничимся лишь некоторыми практическими рецептами их использования, в частности, при вычислении объемов (площадей) тел.

Предположим, необходимо вычислить объем тела (области пространства), форма которой достаточно сложна для того, чтобы это сделать аналитически. В этом случае тело, объем которого нужно вычислить, условно и целиком помещается внутри другого тела, объем которого известен или легко вычисляется – например, квадрат или сферу. Далее случайным образом внутри этого внешнего тела выбираются точки и вычисляется отношение точек, которые попали внутрь меньшего тела (объем которого вычисляется) к общему числу точек. Если общее число точек стремится к бесконечности, то указанное отношение стремится к отношению объемов внутреннего и внешнего тел. На практике вместо случайного выбора точек используют сетку из равномерно распределенных точек. В этом случае говорят о квази-методах Монте-Карло.

Рассмотрим программу для вычисления объема тела, ограниченного поверх-

ностями $\frac{\sqrt{x^2 + y^2}}{R} = \frac{z}{H}$ при $0 \leq z \leq H$ и $x^2 + y^2 + (z - H)^2 = R^2$ при

$H < z \leq R + H$. В данном случае речь идет об ограниченном конусе с радиусом основания R и высотой H , накрытым сверху полусферой (фигура, напоминающая «крем-брюле»). Объем V такой фигуры равен сумме объемов конуса и полусферы, что позволяет записать $V = \frac{1}{3}\pi R^2 H + \frac{2}{3}\pi R^3$.

Попытаемся вычислить этот объем с помощью метода Монте-Карло. Для этого рассмотрим параллелепипед $-R \leq x \leq R$, $-R \leq y \leq R$ и $0 \leq z \leq H + R$, внутри которого полностью помещается конус с полусферой. Объем параллелепипеда равен $V_0 = 4R^2(H + R)$. Если из N точек, выбранных внутри параллелепипеда, n находится внутри конуса, то можем

записать $\lim_{N \rightarrow \infty} \frac{n}{N} = \frac{V}{V_0}$, откуда можем найти $V \approx V_0 \frac{n}{N}$, где соотношение $\frac{n}{N}$ рассчитывается путем генерирования большого числа точек (чем больше точек, тем выше точность результата). Соответствующий программный код приведен в листинге 2.21.

Листинг 2.21. Вычисление объема методом Монте-Карло

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
    //Число pi:
    const double pi=3.1415;
    //Рабочие переменные программы:
    double R,H,V,V0,x,y,z;
    //Число точек N (по каждой из координат)
    //и число внутренних точек n:
    int N=1500,n=0;
    //Ввод параметров R и H:
    cout<<"Enter R = ";
    cin>>R;
    cout<<"Enter H = ";
    cin>>H;
    //Объем параллелепипеда:
    V0=4*R*R*(H+R);
    //Перебор всех точек:
    for(int i=0;i<=N;i++){
        x=2*i*R/N-R;
        for(int j=0;j<=N;j++){
            y=2*j*R/N-R;
            for(int k=0;k<=N;k++){
                z=k*(H+R)/N;
                //Подсчет внутренних точек:
                if( ((sqrt(x*x+y*y)/R<=z/H) && (z<=H)) || ((x*x+y*y+(z-H)*(z-H)<=R*R) && (z>H)) ) n++;
            }
        }
    }
    //Объем тела:
    V=V0*n/pow(N+1,3);
    //Вывод вычисленного и точного значений:
    cout<<"V = "<<V<<" : "<<pi*R*R*H/3+2*pi*pow(R,3)/3<<endl;
    return 0;
}
```

В программе пользователем вводятся геометрические параметры R и H , после чего внешняя область (параллелепипед) «заполняется» сеткой из равноотстоящих точек, для чего используется тройной оператор цикла (по одной индексной

переменной для каждой пространственной размерности). Общее количество точек определяется переменной N (всего точек $(N + 1)^3$). Количество внутренних точек (т.е. точек, попадающих в область, ограниченную конусом и полусферой) записывается в переменную n . Каждый раз при переборе точек проверяется условие $((\text{sqrt}(x^2 + y^2) / R \leq z / H) \&\& (z \leq H)) \vee ((x^2 + y^2 + (z - H)^2 \leq R^2) \&\& (z > H))$, которое, в свою очередь, состоит в том, что должно выполняться одно из двух условий: $(\text{sqrt}(x^2 + y^2) / R \leq z / H) \&\& (z \leq H)$ или $(x^2 + y^2 + (z - H)^2 \leq R^2) \&\& (z > H)$. Условие $(\text{sqrt}(x^2 + y^2) / R \leq z / H) \&\& (z \leq H)$ означает, что точка принадлежит конусу, а условие $(x^2 + y^2 + (z - H)^2 \leq R^2) \&\& (z > H)$ означает принадлежность точки полусфере. Если общее условие выполнено, переменная n увеличивается на единицу. При выводе результата отображается не только вычисленное значение для объема фигуры, но и точное значение. Результат выполнения программы имеет вид (жирным шрифтом выделен ввод пользователя):

```
Enter R = 2
Enter H = 5
V = 37.6237 : 37.698
```

Откровенно говоря, точность вычислений не самая высокая, и это при том, что было взято достаточно большое число точек. Тем не менее методы Монте-Карло доказали свое право на существование и очень часто являются незаменимым инструментом исследования.

Резюме

Верная мысль, если ее часто повторять, теряет силу.

А. Моруа

1. В C++ используется ряд управляющих инструкций, посредством которых в программах реализуются точки ветвления и выполняются циклические повторения однотипных команд. К таким управляющим инструкциям относят условные операторы и операторы цикла.
2. Условный оператор `if()` позволяет выполнять различные блоки команд в зависимости от того, выполняется некоторое условие или нет. Проверяемое условие указывается аргументом оператора `if()`, после чего следует блок команд, выполняемых при справедливости условия. Команды, выполняемые в случае, если условие неверно, указываются после ключевого слова `else`. Допускается использование вложенных словных операторов, что позволяет создавать разветвленную структуру точек ветвления в программе.

3. В условном операторе `switch()` проверяется значение некоторого выражения. Выражение указывается аргументом оператора `switch()`. В зависимости от значения выражения выполняются разные блоки команд. Возможное значение выражения (число или символ) указывается после ключевого слова `case`. В блоке `default` размещаются команды, выполняемые, если ни одно из представленных в `case`-блоках значений выражения не реализовано.
4. Оператор цикла `for()` обычно используется в тех случаях, когда необходимо выполнить определенное число раз предопределенный блок команд. Аргумент оператора `for()` состоит из трех блоков. Блоки разделяются точкой с запятой. Обычно в первом блоке выполняется инициализация индексной переменной, во втором блоке указывается проверяемое условие (оператор цикла выполняется до тех пор, пока условие истинно), в третьем блоке выполняется изменение индексной переменной. Блоки могут быть пустыми. Команды, выполняемые в каждом цикле, размещаются в фигурных скобках после инструкции `for()`.
5. В операторе цикла `while()` проверяется условие, переданное аргументом оператору. Если условие истинно, выполняется блок команд после инструкции `while()` (команды заключаются в фигурные скобки). Условие проверяется после выполнения каждого цикла. Работа оператора прекращается, когда условие становится ложным. Оператор имеет разновидность `do-while()`. Принципиальная разница между этими формами состоит в том, что в первом случае перед выполнением команд оператора проверяется условие, а во втором сначала выполняются команды, а только после этого проверяется условие.
6. В C++ существует инструкция безусловного перехода `goto`, с помощью которой выполняется переход к месту программы, помеченному меткой. С помощью инструкции безусловного перехода также можно реализовывать петли циклов. Однако следует иметь в виду, что обычно это негативно сказывается на читабельности программного кода.

Контрольные вопросы

Давайте переживать неприятности по мере их поступления.

М. Жванецкий

1. Что такое управляющие инструкции и зачем они нужны?
2. Какие условные операторы используются в C++?

3. Когда и как используется условный оператор `if()`? Каковы его особенности?
4. В чем особенности условного оператора `switch()`? Когда и как он используется?
5. Что такое оператор цикла `for()`? В каких случаях он используется, и каковы его особенности?
6. Когда и как используется оператор цикла `while()`? Каковы его особенности?
7. В чем особенности оператора цикла `do-while()` по сравнению с оператором цикла `while()`?
8. Что такое инструкция безусловного перехода? Каков принцип ее использования? Какие преимущества и недостатки использования инструкций безусловного перехода?

Задачи для самостоятельного решения

Задача 1. Написать программу для вычисления косинуса

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}.$$

Аргумент x вводится пользователем с клавиатуры, а граница ряда определяется как константа.

Задача 2. Написать программу для вычисления экспоненты

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

Аргумент x вводится пользователем с клавиатуры, а граница ряда определяется как константа.

Задача 3. Написать программу для вычисления ряда

$$1 + 2x + 3x^2 + 4x^3 + \dots = \sum_{n=0}^{\infty} (n+1)x^n = \frac{1}{(1-x)^2}.$$

Аргумент x (значение $|x| < 1$) вводится пользователем с клавиатуры, а граница ряда определяется как константа.

Задача 4. Написать программу для вычисления ряда

$$1 - 2x + 3x^2 - 4x^3 + \dots = \sum_{n=0}^{\infty} (-1)^n (n+1)x^n = \frac{1}{(1+x)^2}.$$

Аргумент x (значение $|x| < 1$) вводится пользователем с клавиатуры, а граница ряда определяется как константа.

Задача 5. Написать программу для вычисления логарифма

$$\ln(1+x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)}.$$

Аргумент x (значение $|x| < 1$) вводится пользователем с клавиатуры, а граница ряда определяется как константа.

Задача 6. Написать программу для вычисления ряда

$$\frac{\sin(x)}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n+1)!}.$$

Аргумент x вводится пользователем с клавиатуры, а граница ряда определяется как константа.

Задача 7. Написать программу для вычисления факториала числа $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$. Число n вводится пользователем с клавиатуры.

Задача 8. Написать программу для вычисления двойного факториала числа $n!! = n \cdot (n-2) \cdot (n-4) \cdot \dots$. Число n вводится пользователем с клавиатуры.

Задача 9. Написать программу для вычисления произведения

$$\prod_{n=0}^{\infty} \left(1 + \left(\frac{1}{2} \right)^{2n} \right) = 2.$$

Верхняя граница произведения вводится пользователем с клавиатуры.

Задача 10. Написать программу для вычисления произведения

$$\prod_{n=1}^{\infty} \cos \left(\frac{\pi}{2^{n+1}} \right) = \frac{2}{\pi}.$$

Верхняя граница произведения вводится пользователем с клавиатуры.

Задача 11. Написать программу для вычисления времени полета тела, брошенного под углом к горизонту. Начальная скорость тела равна V и направлена под углом α к горизонту. Для вычислений использовать дискретную модель.

Задача 12. Написать программу для вычисления положения тела, брошенного под углом к горизонту, в произвольный момент времени t . Учесть сопротивление воздуха: сила трения пропорциональна скорости тела. Начальная скорость тела равна V и направлена под углом α к горизонту. Для вычислений использовать дискретную модель. В векторном виде уравнение движения имеет вид $m\ddot{\vec{r}} = m\vec{g} - \gamma\dot{\vec{r}}$, где через γ обозначен коэффициент пропорциональности, входящий в выражение для силы сопротивления воздуха $\vec{F} = -\gamma\dot{\vec{r}}$. В проекциях на координатные оси получаем уравнения $m\ddot{x} = -\gamma\dot{x}$ и $m\ddot{y} = -mg - \gamma\dot{y}$. В модели дискретного времени с обозначениями x_n – горизонтальная координата в момент времени t_n , y_n – вертикальная координата в момент времени t_n , v_n – проекция скорости на горизонтальную ось в момент времени t_n , u_n – проекция скорости на вертикальную ось в момент времени t_n , получаем уравнения (Δt – шаг дискретности по времени) $x_{n+1} = x_n + v_n\Delta t$, $y_{n+1} = y_n + u_n\Delta t$, $v_{n+1} = v_n - kv_n\Delta t$ и $u_{n+1} = u_n - g\Delta t - ku_n\Delta t$, где обозначено $k = \gamma/m$. Для сравнения: точное решение имеет вид $x(t) = \frac{V \cos(\alpha)}{k}(1 - \exp(-kt))$, $y(t) = \frac{g/k + V \sin(\alpha)}{k}(1 - \exp(-kt)) - \frac{gt}{k}$.

Задача 13. Написать программу для решения квадратного уравнения, которое имеет общий вид $ax^2 + bx + c = 0$. Параметры a , b и c вводятся пользователем. Использовать аналитические формулы для решений уравнения, при этом учесть различные варианты: например, $a = 0$, отсутствие решений (комплексные решения).

Задача 14. Написать программу для решения уравнения $a \sin(x) + b \cos(x) = c$. Параметры a , b и c вводятся пользователем. Воспользоваться тем, что уравнение сводится к уравнению вида $\sin(x + \alpha) = c/\sqrt{a^2 + b^2}$, где $\cos(\alpha) = a/\sqrt{a^2 + b^2}$ и $\sin(\alpha) = b/\sqrt{a^2 + b^2}$. Предусмотреть ситуацию, когда уравнение решений не имеет (например, при $|c| > \sqrt{a^2 + b^2}$).

Задача 15. Написать программу для вычисления методом последовательных итераций уравнения $x = A \cos(x)$. Параметр A вводится пользователем. Проверить, для каких значений параметра A применим метод последовательных итераций.

Задача 16. Написать программу для вычисления методом последовательных итераций уравнения $x = A \exp(-x)$. Параметр A вводится пользователем. Проверить, для каких значений параметра A применим метод последовательных итераций.

Задача 17. Модифицировать программу «Калькулятор» так, чтобы кроме четырех арифметических операций (сложение, вычитание, умножение и деление), выполнялось еще и возведение в степень.

Задача 18. Написать программу для вычисления значения числа π с помощью метода Монте-Карло. Для этого рассмотреть квадрат с центром в начале координат и длиной ребра 2, в который вписана окружность радиуса 1 с центром в начале координат. Вероятность того, что выбранная наугад точка внутри квадрата попадет внутрь окружности равна отношению площадей окружности и квадрата, т.е. равна $\pi/4$. Площадь квадрата покрывается сеткой из равноотстоящих точек (чем выше плотность точек, тем выше точность вычисления числа π). Отношение точек, попавших внутрь окружности, к общему числу точек стремится (при увеличении общего количества точек к бесконечности) к вероятности попадания случайной точки внутрь окружности, т.е. к числу $\pi/4$. Если координаты точки x и y ($-1 < x < 1$ и $-1 < y < 1$), то условие попадания точки внутрь окружности имеет вид $x^2 + y^2 < 1$.

Задача 19. Написать программу для вычисления методом Монте-Карло площади S тела, ограниченного кривыми $xy = a^2$ и $x + y = \frac{5}{2}a$. Параметр $a > 0$ вводится пользователем. Сравнить результат с точным значением $S = \left(\frac{15}{8} - 2 \ln 2 \right) a^2$.

Задача 20. Написать программу для вычисления методом Монте-Карло объема V тела, ограниченного поверхностями $z = x^2 + y^2$, $y = x^2$, $y = 1$ и $z = 0$. Сравнить с точным значением $V = \frac{88}{105}$.

Глава 3

Указатели, ссылки и массивы

Нам нет необходимости наступать на те же грабли, что уже были.

В.С. Черномырдин

Переменную мы определяли как именованную область памяти. К этой области обращаются по имени, которое и является именем переменной, определяемой в программном коде. Когда в программе объявляется переменная, для этой переменной в памяти выделяется место. Объем выделяемой памяти определяется типом переменной. При присваивании переменной значения это значение записывается в область памяти, выделенную для переменной. Технически при работе с переменной пользователь ничего не знает (да это и не нужно!) о физическом адресе той области, куда записывается значение переменной. Другими словами, вся «кухня» по размещению значения переменной в памяти скрыта от пользователя. С одной стороны, это удобно. Но на практике нередко встречаются ситуации, когда необходимо иметь прямой доступ к областям памяти, содержащим значения переменных. Делается это с помощью указателей.

Под указателем подразумевают переменную, значением которой является адрес памяти. Таким образом, если значением обычной переменной является то, что записано в определенной области памяти, то значением переменной-указателя является адрес области памяти. Поясним разницу между обычной переменной и переменной-указателем, на примере, не имеющем прямого отношения к программированию.

Предположим, мы знакомимся с человеком и вносим в записную книжку его имя и адрес. В данном случае имя нового знакомого – это имя переменной, а его адрес – указатель. В чем разница между переменной (имя знакомого) и указателем (адрес знакомого)? Ведь по известному адресу мы в принципе можем определить, кто проживает в квартире. Зная имя человека, можем, как правило, определить его адрес. Если человек не меняет своего места жительства, разница между указателем и переменной состоит только в способе использования переменной и указателя. Например, если наш новый знакомый – симпатичная девушка и мы хотим пригласить ее в театр, то сделать это можно двумя способами: либо напрямую обратиться к ней (при встрече или по телефону), либо отправить по ее адресу открытку. Первый случай более надежен, поскольку мы точно знаем, кого и куда приглашаем.

Во втором случае, если девушка сменила адрес, возможны недоразумения. С другой стороны, если нас в первую очередь интересует квартира, то очевидно, что для осмотра помещения нам понадобится адрес, причем абсолютно не имеет значения, какая девушка там в данный момент проживает. Другими словами, оперируя в программе с переменными, мы фактически даем инструкции на предмет того, что делать со значениями этих переменных. Указатели позволяют выполнять действия со значениями, записанными по определенному адресу. Если указатель содержит в качестве значения адрес памяти с данными, то говорят, что указатель ссылается на эти данные.

Объявление и использование указателей

Недостаточно знания, необходимо также применение. Недостаточно хотеть, надо и делать.

И. Гете

Поскольку указатель – это переменная, его необходимо объявлять. Как и для обычных переменных, для указателя важен тип данных, к которому он относится. Дело в том, что объем памяти, отводимый под переменную, зависит от ее типа. Хотя формально указатель в качестве значения может иметь любой адрес памяти, от типа хранящихся там данных зависит результат арифметических операций с адресами. Поэтому при объявлении указателей необходимо указывать тип данных, на которые эти указатели могут ссылаться. При объявлении указателя используется оператор «звездочка» *. Оператор указывается перед именем указателя. Во всем остальном способ объявления указателя мало отличается от объявления обычной переменной. Синтаксис объявления указателя следующий:

тип_данных *имя_указателя;

Например, командой `int *p` объявлен указатель `p` на значение целочисленного типа. Причем одновременно можно объявлять как обычные переменные, так и указатели. Командой `int *q, n, *p` объявляется целочисленная переменная `n` и два указателя `q` и `p` на значения целого типа.

Существует две операции, которые приходится часто выполнять при работе с указателями. Во-первых, это определение адреса ячейки по ее имени и, во-вторых, определение значения, записанного по указанному адресу. Для этих целей используются операторы `&` и `*` соответственно. В частности, для того, чтобы определить адрес, по которому записана переменная, необходимо перед ее именем указать оператор `&`. Чтобы по адресу (указателю) определить

значение, перед соответствующим указателем ставим оператор *. В листинге 3.1 приведен пример использования указателя на значение целого типа.

Листинг 3.1. Использование указателя

```
#include <iostream>
using namespace std;
int main() {
    int *q, n, *p;
    n=100;
    p=&n;
    q=p;
    (*p)++;
    cout<<*q<<"\n";
    cout<<n<<"\n";
    cout<<p<<"\n";
    return 0;
}
```

Сначала объявляется целочисленная переменная `n` и два указателя `q` и `p`. Командой `n=100` переменной `n` присваивается значение 100. Далее с помощью команды `p=&n` в переменную-указатель `p` записывается адрес области памяти, в которой хранится значение переменной `n`. Обращаем внимание читателя на то, что значением переменной `p` является именно адрес! Командой `q=p` переменной `q` в качестве значения присвоен адрес, являющийся значением переменной `p`. В данном случае использовано то свойство, что при совпадении типов один указатель можно присвоить в качестве значения другому указателю. В результате указатель `p` и указатель `q` ссылаются на одну и ту же область памяти. Командой `(*p)++` значение, записанное по адресу `p`, увеличивается на единицу. Напомним, что инструкция `*p` означает ссылку на то значение, что записано по адресу `p`. Команду `(*p)++` следует понимать так: взять значение, записанное по адресу `p`, и увеличить его на единицу (поскольку оператор `++` имеет более высокий приоритет, чем оператор `*`, используем скобки). У этой команды более серьезные последствия, чем может показаться на первый взгляд. Прежде всего, в результате на единицу увеличивается значение переменной `n`. Действительно, поскольку адрес `p` определялся командой `p=&n`, то фактически, по определению, это адрес области памяти, где хранится значение переменной `n`. Поскольку соответствующее значение увеличено на единицу, это означает, что на единицу увеличилось значение переменной `n`. Далее, указатель `q` ссылается на эту же область памяти. Поэтому результатом команды `*q` (значение, записанное по адресу `q`) будет увеличенное на единицу значение переменной `n`, т.е. число 101. В конце выполнения программы на экран выводятся значения `*q`, `n` и `p`. Первые два значения, в силу указанных выше причин, равны 101.

В качестве значения `p` выводится адрес области памяти – это число в шестнадцатеричном представлении. Результат может быть, например, таким:

```
101
101
0012FF78
```

В данном случае измениться может только третья строка вывода результатов программы, первые две будут неизменными.

Адресная арифметика и сравнение указателей

У нас нет денег, поэтому нам приходится думать.

Э. Резерфорд

О том, что один указатель можно присваивать в качестве значения другому указателю, выше уже упоминалось. Помимо этого, указатели можно сравнивать, а также к указателям можно применять операции сложения и вычитания. И если при присваивании указателей результат достаточно очевиден, то предугадать последствия других операций не так уж легко.

Сначала рассмотрим арифметические операции, допустимые при работе с указателями. В частности, к указателям можно прибавлять и отнимать целые числа, а также допускается вычитание указателей. Другие арифметические операции по отношению к указателям недопустимы.

Если к указателю прибавить (отнять) целое число, получим новый адрес. Этот новый адрес ссылается на ячейку области памяти, в которую записываются элементы базового типа. Новый адрес определяется из условия, что между исходной ячейкой и той, что определяется новым адресом, могли разместиться элементы базового типа в количестве, равном прибавляемому (отнимаемому) числу. При прибавлении чисел адрес увеличивается, при отнимании – уменьшается. Если элемент данных в памяти занимает больше одной ячейки, адресом считается адрес первой ячейки области, в которую записан элемент. Например, если указатель `p` в качестве значения имеет адрес 3000, а для записи базового типа необходимо 4 байта, то в результате операции `p+3` получим адрес 3012. Причина проста: адрес `p+3` должен быть таким, чтобы между ним и адресом `p` разместилось ровно 3 элемента базового типа, каждый из которых занимает 4 байта (всего 12 байт). Аналогично, результатом операции `p--` будет адрес 2196.

Результатом вычитания двух адресов (указателей) является целое число, указывающее, сколько элементов базового типа может разместиться меж-

ду ячеек с соответствующими адресами. Например, если указатель p имеет в качестве значения адрес 3000, а указатель q – значение 2000 (при условии, что базовый тип занимает 4 байта), то результатом команды $p-q$ будет число 250 (результат определяется как $(3000-2000)/4=250$). Если от большего по абсолютной величине адреса отнимается меньший адрес – число будет отрицательным.

В данном случае мы естественным образом пришли к процедуре сравнения адресов (указателей). Сравнение выполняется так же, как и сравнение обычных переменных. Поскольку адрес как таковой является не чем иным, как числовым значением, то большим считается тот адрес, числовое значение которого больше. Сравнение указателей широко применяется при работе с массивами. О массивах, в том числе и массивах указателей, речь пойдет в следующих главах.

Многоуровневая адресация

Мы будем проводить иностранную политику иностранными руками.

Дж. Буш (младший)

Внимательный читатель наверняка обратил внимание на то замечательное обстоятельство, что указатель – это переменная. А поскольку это переменная, то ее значение должно быть где-то записано. Таким образом, если мы создаем переменную и ссылаемся на нее посредством указателя, то адрес соответствующей ячейки памяти (значение указателя) в свою очередь также является значением какой-то ячейки. У этой ячейки, очевидно, есть адрес. И тут возникает вопрос: а можно ли этот адрес присвоить в качестве значения указателю? Другими словами, можно ли создать указатель на указатель? Оказывается, можно!

Наша задача состоит в том, чтобы создать указатель, который будет ссылаться на переменную-указатель, которая ссылается на обычную переменную. Ситуацию иллюстрирует рис. 3.1.

В верхней части рисунка представлена схема стандартной, одноуровневой адресации: переменная-указатель через свое значение-адрес ссылается на обычную переменную. Нижняя часть рисунка иллюстрирует двухуровневую адресацию: на переменную ссылается указатель, а на этот указатель, в свою очередь, ссылается еще один указатель. Значением второго указателя является адрес ячейки, в которую записан адрес исходной переменной. Легко понять, что двухуровневая адресация не является верхом совершенства

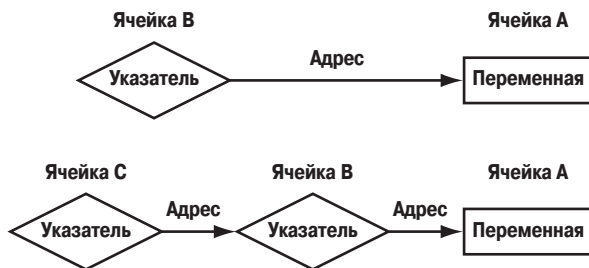


Рис. 3.1. Одноуровневая и двухуровневая адресация

и по аналогии может быть трехуровневая, четырехуровневая и т.д. адресация. Однако сразу отметим, что адресация уровня больше двух на практике используется редко.

Для объявления переменной-указателя на указатель перед именем этой переменной указывают два оператора *. Каждый дополнительный уровень адресации означает наличие дополнительного оператора * в объявлении переменной-указателя. Например, командой `int **q` объявляется переменная-указатель на указатель на целочисленное значение. Пример использования двухуровневой адресации представлен в листинге 3.2.

Листинг 3.2. Двухуровневая адресация

```

#include <iostream>
using namespace std;
int main(){
    int n,*p,**q;
    p=&n;
    q=&p;
    n=100;
    (*p)+=5;
    (**q)--;
    cout<<n<<"\n";
    cout<<*p<<"\n";
    cout<<**q<<"\n";
    cout<<p<<"\n";
    cout<<q<<"\n";
    return 0;
}
    
```

В программе командой `int n,*p,**q` объявляется целочисленная переменная `n` и два указателя: указатель `p` на переменную типа `int` и указатель `q` на указатель для указателя на переменную типа `int`. В качестве значения указателю `p` командой `p=&n` присваивается адрес ячейки `n`, а указатель `q` получает в качестве значения адрес ячейки, в которую записано значение

переменной-указателя `p` (команда `q=&p`). Обращаем внимание читателя на то, что обе указанные операции выполнены до того, как переменной `n` присвоено значение. Такая ситуация является корректной, поскольку память для переменной `n` выделяется при ее объявлении. Следовательно, даже если переменной значение еще не присвоено, адрес области памяти, куда это значение будет заноситься, доступен.

После выполнения команд `n=100` и `(*p)+=5` значение переменной `n` становится равным 105. После этого данное значение уменьшается на единицу – вследствие выполнения команды `(**q)--`. В данном случае мы воспользовались тем, что `*q` есть значение, записанное по адресу `q` (по адресу `q` записан указатель `p`), а `**q` – это значение, записанное по адресу `*q` (т.е. значение, записанное по адресу `p`, а это переменная `n`). Результат выполнения команды может иметь следующий вид:

```
104
104
104
0012FF7C
0012FF78
```

Неизменными являются первые три строки вывода результата. Две последние строки – значения (адреса) указателей. Они могут отличаться от тех, что приведены выше.

Хотя система двухуровневой адресации может показаться слишком замысловатой, она имеет конкретное практическое применение, особенно при работе с двумерными массивами.

Знакомство со ссылками

*Надо, чтоб и победа была, и чтоб
без войны. Дипломатия, пони-
маешь.*

Б. Ельцин

В C++ существует неявная форма указателя – ссылка. По своей, что называется, природе, ссылка действительно схожа с указателем. Однако, в отличие от указателя, при работе со ссылкой пользователь не получает прямого доступа к адресу ячейки памяти. В некотором смысле ссылка – это указатель, но только со скрытым адресом. Способ объявления и использования ссылок в программе полностью отличается от способов работы с указателями.

Обычно ссылки используются при передаче аргументов функциям, при возвращении значений функций через ссылки, а также при создании псевдонимов.

мов объектов. В последнем случае речь идет о независимой ссылке. На ней остановимся подробнее.

Независимая ссылка – это фактически еще одно название для переменной в программе. Значение ссылки – это значение переменной, на которую ссылка выполнена (в этом смысле она отличается от указателя, значением которого является адрес ячейки, на которую ссылается указатель). При объявлении ссылки необходимо сразу указывать, на какую переменную она ссылается (в этом случае говорят об инициализации ссылки). Ссылка инициализируется только один раз. При объявлении (и инициализации) ссылки перед ее именем указывается оператор `&`. В дальнейшем имя ссылки служит еще одним альтернативным названием для переменной, на которую выполнена ссылка. В листинге 3.3 приведен пример программы с использованием ссылки.

Листинг 3.3. Использование ссылок

```
#include <iostream>
using namespace std;
int main(){
    int n;
    int &copy=n;
    copy=100;
    n++;
    cout<<n<<"\n";
    cout<<copy<<"\n";
    return 0;
}
```

В программе объявляется целочисленная переменная `n`. Командой `int ©=n` объявляется независимая ссылка `copy`, значением которой указана переменная `n`. Это означает, что в дальнейшем `n` и `copy` можно использовать как независимые названия для одной и той же переменной. В частности, после выполнения команды `copy=100` такое же значение получит переменная `n`. Поэтому вполне корректна и следующая команда `n++`. После ввода на экран значений `n` и `copy` убеждаемся, что в обоих случаях значение равно 101.

Убедиться в том, что независимая ссылка на практике является всего лишь альтернативным способом обращения к переменной, можно на следующем простом примере. В листинге 3.4 представлен код программы, в которой одновременно используются и ссылки, и указатели. Причем создается указатель как на переменную, так и на ссылку на эту переменную (хотя существует рекомендация не использовать указатели на ссылки, большинство компиляторов поддерживают такой механизм).

Листинг 3.4. Использование ссылок и указателей

```

#include <iostream>
using namespace std;
int main(){
    int n,*p,*q;
    int &copy=n;
    p=&n;
    copy=100;
    (*p)/=10;
    q=&copy;
    n++;
    cout<<n<<"\n";
    cout<<copy<<"\n";
    cout<<*p<<"\n";
    cout<<*q<<"\n";
    cout<<p<<"\n";
    cout<<q<<"\n";
    return 0;
}

```

По существу речь идет о дополнении кода из листинга 3.3 несколькими командами, относящимся к созданию указателей. Так, указатели `p` и `q` командами `p=&n` и `q=©` инициализируются со значениями-адресами переменной `n` и ссылки `copy` соответственно. Как и в предыдущем случае, ссылка `copy` (а, значит, и переменная `n`) с помощью команды `copy=100` получает значение 100. После этого командой `(*p)/=10` значение переменной `n` (и ссылки `copy`) уменьшается в 10 раз. После команды `n++` это значение становится равным 11. Результат выполнения программы имеет следующий вид:

```

11
11
11
11
0012FF7C
0012FF7C

```

Первые четыре строки однозначно содержат значение 11 (это одно и то же значение, но полученное через разные механизмы: через переменную `n`, по ссылке `copy`, через указатель `p` и через указатель `q`). Последние две строки могут отличаться от тех, что приведены выше, но в любом случае они совпадают между собой. Это адрес переменной `n` и ссылки `copy`. Совпадение адресов свидетельствует о том, что на самом деле речь идет об одной и той же переменной.

О ссылках речь еще будет идти. Они исключительно важны при работе с функциями и составляют основу одного из двух фундаментальных механизмов передачи функциям аргументов. Что касается независимых ссылок,

то их широкое использование на практике остается под вопросом, поскольку наличие псевдонимов у переменных, как правило, свидетельствует о плохой организации программы. Тем не менее, при работе с классами ссылки бывают весьма полезными.

Статические одномерные массивы

Чужих меж нами нет, мы все друг другу братья под сакурой в цвету...

Из к/ф «Гибель Империи»

Достаточно часто приходится иметь дело с наборами данных одного типа. Обычно такие данные в программе реализуют в виде массива. Под массивом подразумевают совокупность переменных одного типа, объединенных общим именем. Переменные, входящие в состав массива, называются элементами массива. Доступ к элементам массива осуществляется путем индексирования. Размерность массива определяется количеством индексов, необходимых для однозначного определения элемента массива. Массивы бывают статические (размер известен при компиляции программы) и динамические (размер определяется при выполнении программы). Динамические массивы обсуждаются в главе 5. Сейчас же мы остановимся более детально на том, как реализуются статические массивы в C++.

Под одномерным подразумевают массив, для индексации элементов которого используют один индекс. Как и в случае с обычной переменной, перед использованием массива его следует объявить. Объявление массива выполняется следующим образом: указывается тип данных, к которым принадлежат элементы массива, имя массива, а также его размер (количество элементов массива). Размер массива указывается в квадратных скобках сразу после имени массива. Например, командой `int m[10]` объявляется целочисленный массив с именем `m`, который состоит из 10 элементов. Размер массива задается числовым литералом или числовой константой. Размер массива должен быть известен на момент компиляции программы и не изменяется в процессе ее выполнения. Обращение к элементу массива выполняется через имя массива с индексом элемента в квадратных скобках. При этом следует помнить, что индексация элементов в C++ начинается с нуля. Таким образом, первым элементом обозначенного выше массива является `m[0]`, а последним, десятым – элемент `m[9]`. Эта особенность языка C++ становится особенно важной с учетом того, что при компиляции и выполнении программ проверка на предмет выхода за пределы массива не выполняется. Такая проверка в полной мере ложится на плечи программиста – язык профессионалов требует профессионального отношения.

В листинге 3.5 приведен код программы, которой массив из 10 элементов заполняется случайными числами, а потом эти значения выводятся в строку на экране.

Листинг 3.5. Создание одномерного массива

```
#include<iostream>
using namespace std;
int main(){
    int n[10];
    for(int i=0;i<10;i++){
        n[i]=rand() % 10;
        cout<<n[i]<<" ";
    }
    cout<<"\n";
    return 0;
}
```

В программе использован оператор цикла, индексная переменная *i* которого пробегает значения от 0 до 9 включительно. Для генерации случайного числа использована функция `rand()`, после чего вычисляется остаток от деления этого числа на 10 (команда `rand() % 10`). Полученное значение присваивается элементу массива и выводится на экран командой `cout<<n[i]<<" "`.

Важной особенностью языка C++ является то, что ячейки памяти, в которые заносятся значения элементов массива, размещены рядом, то есть являются смежными. Эта особенность является достаточно полезной и может успешно использоваться на практике. Об этом речь еще будет идти далее. С этой же особенностью языка C++ связана потенциальная опасность использования массивов. Поскольку проверки на предмет выхода за пределы массива в C++ нет, неверно указанный индекс элемента массива приводит к тому, что выполняется обращение к одной из смежных ячеек за пределами массива. Такая ситуация как ошибочная не идентифицируется, при этом последствия могут быть достаточно трагичными. Если речь идет о считывании значения из ячейки за пределами массива, это, скорее всего, скажется только на выполнении программы. Но вот если в такую ячейку значение записывается, то дело может закончиться не только ошибкой в выполнении программы, но и крахом всей операционной системы.

Указатель на массив

*Породниться родством по душе,
а не по крови, может один только человек.*

Н.В. Гоголь

Еще одна особенность C++ связана с тем, что имя массива (без индексов) является указателем на первый элемент массива. Например, если массив создается командой `int n[10]`, то имя массива `n` является указателем (адресом) на первый элемент массива `n[0]`. В принципе, адрес этого эле-

мента можно получить и стандартными методами, как для обычной переменной с помощью команды `&n[0]`. Однако профессионалы предпочитают использовать более простой синтаксис, основанный на имени массива.

В самой по себе ссылке на первый элемент проку было бы мало, если бы не то принципиальное обстоятельство, что элементы массива размещаются в смежных ячейках. Поэтому, зная адрес первого элемента и количество элементов в массиве, получаем доступ ко всему массиву. Хотя доступ может быть получен и через имя массива и индекс элемента, арифметические операции с адресами выполняются быстрее, по сравнению с индексированием массива. Кроме того, указатель, который ссылается на массив, можно индексировать точно так же, как если бы это было имя массива. Пример использования указателей на массивы приведен в листинге 3.6.

Листинг 3.6. Указатель на массив

```
#include <iostream>
using namespace std;
int main() {
    int n[10], *p;
    p=n;
    for(int i=0; i<10; i++) {
        p[i]=10-i;
        cout<<* (p+i)<<" ";
    }
    cout<<"\n";
    return 0;
}
```

В программе командой `int n[10], *p` объявляется целочисленный массив `n` из 10 элементов и указатель `p` на целое число. Далее командой `p=n` указателю в качестве значения присваивается адрес первого элемента массива. Заполнение элементов массива осуществляется в операторе цикла `for(int i=0; i<10; i++)` в рамках которого командой `p[i]=10-i` выполняется заполнение элементов массива (числа от 10 до 1) и вывод этих значений на экран (команда `cout<<* (p+i)<<" "`). Обращаем внимание читателя, что при заполнении массива значениями ссылка на элементы выполнялась через индексацию указателя на массив в формате `p[i]`. Вывод значения элемента массива осуществлялся методами адресной арифметики. В частности, адрес ячейки элемента массива `n[i]` может быть получен в результате выполнения команды `p+i` (где `p` – указатель на первый элемент массива `n[0]`). Значение элемента массива `n[i]` может быть вычислено как `*(p+i)`. Именно эта инструкция использовалась в команде вывода значения элемента массива на экран. Таким образом, результат выполнения рассмотренной программы будет иметь вид:

```
10 9 8 7 6 5 4 3 2 1
```

В известном смысле использование в приведенном выше коде указателя `p` является лишним, поскольку, как отмечалось, имя массива само является указателем на первый элемент массива. В этом смысле вполне допустимым является приведенный в листинге 3.7 программный код.

Листинг 3.7. Указатель на массив

```
#include <iostream>
using namespace std;
int main() {
    int n[10];
    for(int i=0; i<10; i++) {
        *(n+i)=10-i;
        cout<<n[i]<<"\n";
    }
    return 0;
}
```

В программе, как и в предыдущем случае, заполняются элементы целочисленного массива `n`. Значения массива в столбик выводятся на экран. При заполнении элементов в операторе цикла вместо традиционной ссылки `n[i]` на элементы массива использовалось выражение `*(n+i)`. Здесь как раз использовано то обстоятельство, что имя массива `n` является указателем на элемент `n[0]`.

Двумерные массивы

Дивные звери с двумя хвостами...

Из к/ф «Следопыт»

Размерность массива может быть больше единицы (напомним, что размерность массива определяется количеством индексов, с помощью которых реализуется доступ к элементу массива). В таком случае говорят о многомерных массивах. Объявление многомерного массива выполняется так же просто, как и объявление одномерного массива, с той лишь разницей, что теперь для массива указывается размер по каждому из индексов. Для каждого индекса используется собственная пара квадратных скобок. При объявлении массива размер массива по соответствующему индексу также указывается в отдельных квадратных скобках. Среди многомерных массивов самым простым является двумерный массив. В известном смысле двумерный массив – это массив из одномерных массивов. Например, инструкцией `double n[4][5]` объявляется двумерный массив действительных чисел двойной точности размером 4×5 . Как и ранее, чтобы обратиться к отдельному элементу массива, необходимо после имени массива указать индексы

этого элемента (каждый индекс в отдельных квадратных скобках). Индексация по каждому индексу начинается с нуля. В листинге 3.8 приведен код программы, которой двумерный массив заполняется случайными числами с последующим выводом этих значений на экран.

Листинг 3.8. Заполнение двумерного массива случайными числами

```
#include<iostream>
using namespace std;
int main(){
    int n[4][5];
    for(int i=0;i<4;i++){
        for(int j=0;j<5;j++){
            n[i][j]=rand() % 10;
            cout<<n[i][j]<<" ";
        }
        cout<<"\n";
    }
    return 0;
}
```

Значения массива выводятся построчно в соответствии со структурой массива. Первый индекс массива определяет строку, второй индекс определяет столбец в этой строке. Так, элемент `n[1][3]` находится на пересечении второй строки и четвертого столбца.

Эффективность работы с двумерными (и многомерными) массивами напрямую связана с тем, как такие массивы технически реализуются в области памяти. Другими словами, для понимания основных механизмов в использовании массивов необходимо иметь четкое представление о природе массивов в C++. В частности, при работе с двумерными массивами с успехом могут использоваться указатели. Однако в данном случае принципиально важно помнить о том, что такое на самом деле двумерный массив и как для его элементов в памяти выделяется место. Выше отмечалось, что двумерный массив в C++ – это обычный массив, элементами которого, в свою очередь, являются массивы. Размерность первого, «базового» массива при объявлении определяется числом в первых квадратных скобках, а размерность массивов-элементов – число во вторых квадратных скобках. Таким образом, формально указав имя массива и только первый индекс элемента, получаем ссылку на соответствующую строку двумерного массива. Более точно, название двумерного массива с одним лишь первым индексом есть не что иное, как указатель на первый элемент соответствующей строки массива. Это важное замечание для понимания принципов использования указателей с двумерными (многомерными) массивами. Пример использования указателей при работе с двумерными массивами приведен в листинге 3.9.

Листинг 3.9. Указатели и двумерные массивы

```
#include <iostream>
using namespace std;
int main(){
    int n[4][5], *p;
    for(int i=0; i<4; i++){
        p=n[i];
        for(int j=0; j<5; j++){
            *(p+j)=5*i+j+1;
            printf("%4d", n[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

В программе командой `int n[4][5], *p` объявляется двумерный целочисленный массив `n` и указатель `p` на целое число. Далее в рамках оператора цикла `for (int i=0; i<4; i++)` выполняется команда `p=n[i]`, которой указателю `p` в качестве значения присваивается адрес первого элемента строки двумерного массива с номером `i`. При фиксированном индексе строки `i` выполняется цикл `for (int j=0; j<5; j++)`, которым перебираются все элементы массива в данной строке. Для каждой пары значений индексов `i` и `j` выполняются команды `*(p+j)=5*i+j+1` и `printf("%4d", n[i][j])`. Первая из команд предназначена для записи значения соответствующего элемента массива. Ссылка на элемент выполнена через указатель `p` с использованием адресной арифметики. Поскольку `p` есть ссылка на первый элемент в строке с номером `i`, то результатом инструкции `p+j` является адрес элемента с индексом `j` в этой строке, т.е. адрес элемента `n[i][j]`. Значение элемента получаем с помощью оператора `*`. Командой `printf("%4d", n[i][j])` значение выводится на экран. В данном случае массив заполняется числами от 1 до 20. Чтобы выдержать форматирование выводимых данных (числа должны располагаться строго по вертикали и горизонтали, без сдвигов из-за различной ширины чисел), использована функция `printf()`. Первый ее аргумент `"%4d"` означает, что для вывода каждого числа отводится 4 позиции, второй аргумент – выводимое на экран значение. Командой `printf("\n")` (см. листинг 3.9) осуществляется переход к новой строке. Результат выполнения программы будет следующим:

```
1  2  3  4  5
6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
```

Как и при работе с одномерными массивами, в двумерных массивах само по себе имя массива является ссылкой на первый элемент массива. Одна-

ко в случае с двумерными массивами есть одна существенная особенность. Если, например, объявить целочисленный указатель и затем попытаться присвоить ему в качестве значения имя массива, при компиляции программы получим сообщение об ошибке. Проблема связана с несоответствием типов. Ранее при знакомстве с указателями отмечалось, что важной является информация о типе указателя (имеется в виду тип данных, на которые может ссылаться указатель). Эта информация необходима в первую очередь для корректного выполнения арифметических операций с адресами. Зададимся вопросом: к какому типу относится указатель, определяемый именем массива? Мы знаем, что технически двумерный массив – это массив массивов. Имя одномерного массива – это адрес первого элемента. Первым элементом в двумерном массиве является строка с нулевым индексом. С точки зрения языка C++ эта строка есть массив. Поэтому имя двумерного массива является указателем со значением-адресом первого элемента в первой строке (элемент с двумя нулевыми индексами, в рассмотренном примере это `n[0][0]`), но вот тип этого указателя – массив (целых чисел для случая целочисленного двумерного массива).

Корректными с учетом всего вышеозначенного будут, например, такие команды: `int n[4][5], *q` для инициализации двумерного массива и указателя `q`, а присваивание значения указателю `q` реализуем командой `q=(int *)n`, в которой использована инструкция `(int *)` явного приведения типов. С ее помощью значение указателя `n` приводится к типу «указатель на целое число».

При работе с многомерными массивами следует иметь в виду, что такие структуры занимают достаточно много места в памяти. Если, например, на запись одного элемента массива отводится 4 байта, то одномерный массив из 100 элементов занимает 400 байт, а двумерный массив – 40000 байт. С увеличением размерности массива размер увеличивается в геометрической прогрессии, поэтому на практике массивы размерности больше второй используются редко.

Инициализация массивов

Мы вам ничего не позволим показывать, мы вам сами все покажем.

Из к/ф «Гараж»

В C++ существует возможность инициализации массивов при их объявлении. Для инициализации одномерного массива после имени массива и размера через оператор присваивания указывается список со значениями элементов. Список заключается в фигурные скобки, а сами значения раз-

деляются запятыми. Синтаксис инициализации одномерного массива при объявлении имеет вид:

```
тип имя_массива [размер] = {значение1, значение2, ...};
```

Например, командой `int n[4] = {2, 4, 6, 3}` инициализируется массив `n` из 4 целочисленных элементов, значения которых равны соответственно 2, 4, 6 и 3. Если массив инициализируется при объявлении, указывать размер массива необязательно (хотя квадратные скобки должны присутствовать). В этом случае размер массива определяется автоматически по числу значений в списке. В этом отношении корректной является команда `int n[] = {2, 4, 6, 3}`. При этом массив автоматически получает размерность 4 (по числу значений в списке), его элемент `n[0]` получает значение 2, элемент `n[1]` получает значение 4 и т.д.

Практически так же инициализируются двумерные массивы. Присваивание значений элементам массива при этом выполняется построчно, т.е. сначала заполняются элементы первой строки двумерного массива, затем элементы второй строки массива и т.д. Часто для удобства значения элементов каждой строки заключаются в фигурные скобки. Например, инициализация двумерного массива может выглядеть так:

```
double numbers[3][2] {1.1, 3.2,
                      8.3, 5.4,
                      9.5, 2.6};
```

Этот же массив с такими же начальными значениями мог быть объявлен в виде

```
double numbers[3][2] {{1.1, 3.2},
                      {8.3, 5.4},
                      {9.5, 2.6}};
```

В обоих случаях элемент массива `numbers[0][0]` получает значение 1.1, элемент `numbers[0][1]` получает значение 3.2, элемент `numbers[1][0]` получает значение 8.3 и т.д.

При инициализации многомерных массивов можно первый размер не указывать. В этом случае он определяется автоматически. Например, объявление двумерного массива может выглядеть так:

```
double numbers[][2] {{1.1, 3.2},
                     {8.3, 5.4},
                     {9.5, 2.6}};
```

При инициализации массивов, если явно указан их размер, недостающие элементы заполняются нулями. Например, при объявлении массива командой `int n[4] = {2, 4}` элемент `n[0]` получает значение 2, элемент `n[1]` получает значение 4, а элементы `n[2]` и `n[3]` будут заполнены нулями.

Такая же ситуация с многомерными и, более конкретно, двумерными массивами. Однако здесь следует проявлять некоторую осторожность. В качестве иллюстрации приведем два примера объявления и инициализации массива. В первом случае массив объявляется как

```
int n[3][4]{{1,2},
            {5,6,7},
            {9,10}};
```

Во втором случае массив объявляется (и инициализируется) в виде

```
int n[3][4]{1,2,
            5,6,7,
            9,10};
```

Разница состоит лишь в использовании внутренних фигурных скобок в списке значений элементов массива. Однако массивы инициализируются при этом совершенно по-разному. В первом случае получаем массив

```
1  2  0  0
5  6  7  0
9 10  0  0
```

Во втором случае массив будет иметь вид

```
1  2  5  6
7  9 10  0
0  0  0  0
```

Дело в том, что в первом случае значения указывались разбитыми по строкам с помощью фигурных скобок. Поэтому если для какой-то строки количество явно указанных значений меньше размерности строки, недостающие элементы заполняются нулями. Во втором случае внутренние фигурные скобки не использовались, поэтому последовательно заполняются начальные строки массива, а недостающие значения принимаются равными нулю.

Используя перечисленные выше особенности инициализации массива, легко понять, что для инициализации двумерного массива с нулевыми значениями достаточно использовать команду вида `int n[3][4]{0}`.

Массивы символов

*Словами вера лишь жива,
Как можно отрицать слова.*

И. Гете

Как уже отмечалось, в C++ не существует встроенного типа для текстовых данных. Текстовые строки реализуются в виде массивов символов либо в виде объектов класса `string`. Работа с объектами этого класса обсуждается в приложениях. Особенности работы со строками, реализованными

в виде символьных массивов, описаны в главе 5. Здесь кратко остановимся на основных моментах, которые характерны для массивов символов.

По большому счету массив символов мало чем отличается от массивов иных типов. Главная особенность связана с тем, что при объявлении массива из символов необходимо зарезервировать достаточно места для того, чтобы в такой массив можно было записывать строки разной длины. Другими словами, при реализации строк в виде массивов существует принципиальное ограничение на длину строки. Такое ограничение существует и при использовании статических массивов других типов. Однако там эта проблема не столь актуальна. Поскольку при работе с символьными массивами речь идет о представлении типа данных, а не единичного объекта, необходимо предусмотреть возможность частого изменения данных. Принципиальным показателем при работе с текстовыми данными является длина строки. Поскольку в символьном массиве каждый символ строки соответствует элементу массива, длина строки напрямую имеет отношение к размеру символьного массива. С проблемой ограниченности размера строк связана еще одна проблема. Даже если размер массива достаточно велик для того, чтобы записывать в него строковые значения, необходим индикатор, который позволял бы определить, где в массиве записана полезная информация, а где начинается неинформативный «хвост». В качестве такого индикатора используют специальный символ `'\0'`, который называется нуль-символом.

Таким образом, строковая переменная реализуется в программе в виде массива символов. Признаком окончания строки является нуль-символ.

Чтобы вписать в массив строку, необходимо, чтобы размер массива по крайней мере на единицу превышал количество символов в строке. Этот дополнительный элемент необходим для записи нуль-символа `'\0'` окончания строки.

Объявляются массивы символов, как и прочие массивы: указывается тип элементов массива (для символьных массивов это `char`), название и размер массива. Пример объявления символьного массива:

```
char str[80];
```

В данном случае объявлен массив из 80 символов. При этом в такой массив можно записать строку с максимальной длиной в 79 символов. Инициализироваться символьные массивы могут так же, как и, например, числовые: после имени и размера массива указывается знак равенства и в фигурных скобках список символов, которые являются значениями элементов массива. Однако существует еще один более удобный способ инициализации символьного массива: вместо списка символов указывается в двойных кавычках текстовая строка. Далее в листинге 3.10 приведен пример инициализации символьных массивов:

Листинг 3.10. Инициализация символьного массива

```
#include <iostream>
using namespace std;
int main(){
    char str1[20]="hello";
    char str2[20]={'h','e','l','l','o','\0'};
    cout<<str1<<"\n";
    cout<<str2<<"\n";
    return 0;
}
```

Объявление символьных массивов с одновременной инициализацией выполняется командами `char str1[20]="hello"` и `char str2[20]={'h','e','l','l','o','\0'}`. В обоих случаях объявляется массив из 20 символов. Инициализация в первом случае выполняется путем указания текстового значения (значение заключено в двойные кавычки). Во втором случае это же значение передается в виде заключенного в фигурные скобки списка, причем в явном виде указывается нуль-символ окончания строки `'\0'`. При инициализации массива с помощью текстового литерала нуль-символ добавляется автоматически.

Для вывода значений символьного массива на экран его имя указывается справа от оператора вывода (команды `cout<<str1<<"\n"` и `cout<<str2<<"\n"`).

Ранее отмечалось, что если массив инициализируется при объявлении, размер массива указывать не обязательно. Рассмотрим два следующих способа инициализации символьного массива:

```
char str1[]="hello";
char str2[]={'h','e','l','l','o'};
```

Формально и в том, и в другом случае значением массива является одна и та же текстовая строка `hello`. Однако формируются массивы по-разному. Массив, инициализируемый командой `char str1[]="hello"`, состоит из 6 элементов (хотя букв в слове `hello` только 5). Как уже отмечалось, если символьный массив инициализируется со значением-строкой (значение в двойных кавычках), то в массив в качестве элементов последовательно заносятся все символы строки и еще один символ в конце массива – символ окончания строки, т.е. нуль-символ `'\0'`.

При инициализации символьного массива командой `char str2[]={'h','e','l','l','o'}` ничего подобного не происходит. В этом случае массив состоит из 5 элементов – в соответствии с количеством символьных элементов в списке значений массива.

Массивы указателей

*Все что видим мы – видимость только одна,
Далеко от поверхности мира до дна.
Полагай несущественным явное в мире,
Ибо тайная сущность вещей не видна.*

Омар Хайям

Элементами массива могут быть указатели. В этом отношении указатели не уступают прочим типам данных. Синтаксис объявления массива указателей выглядит следующим образом:

```
тип *имя_массива[размер];
```

Например, чтобы объявить массив из 5 указателей на тип `double`, используем команду `double *pntr[5]`. При работе с массивом `pntr` следует помнить, что его элементы – указатели. Поэтому, например, процедура присваивания элементам массива значений и вывод их на экран может выглядеть следующим образом:

```
double *pntr[5];
double x=3.0;
pntr[0]=&x;
cout<<"x= "<<*pntr[0]<<"\n";
```

В данном случае значение присваивается только первому элементу массива. А именно, командой `double x=3.0` инициализируется переменная типа `double`, после чего командой `pntr[0]=&x` адрес этой переменной присваивается в качестве значения элементу `pntr[0]`. Поскольку значением элемента `pntr[0]` является указатель на переменную, то, чтобы получить доступ к значению этой переменной, необходимо использовать инструкцию вида `*pntr[0]`. Более того, если впоследствии изменится значение переменной `x`, то изменится и результат инструкции `*pntr[0]`. При этом значение элемента массива `pntr[0]` остается неизменным: ссылка на переменную не меняется, но меняется значение переменной.

Массивы указателей – конструкции интересные и весьма полезные. Например, с помощью массива указателей можно создавать массивы с переменной размерностью: в двумерном массиве каждая строка содержит разное число элементов. Рассмотрим следующий фрагмент кода:

```
int *n[3];
int n1[2]={1,2};
int n2[4]={3,4,5,6};
int n3[3]={7,8,9};
n[0]=n1;
```

```
n[1]=n2;
n[2]=n3;
cout<<n[1][2]<<"\n";
```

В результате выполнения приведенной последовательности команд на экране появится число 5. Поясним это более детально.

Первой командой `int *n[3]` объявляется массив указателей целочисленного типа. В массиве три элемента. Далее, командами `int n1[2]={1,2}`, `int n2[4]={3,4,5,6}` и `int n3[3]={7,8,9}` последовательно инициализируются три целочисленных массива разного размера: два, четыре и три элемента соответственно. Напомним, что имена этих массивов являются указателями на первые их элементы. Другими словами, `n1` является, например, указателем на первый элемент `n1[0]` данного массива (а через этот элемент получаем доступ ко всем прочим элементам массива). Поэтому нет ничего удивительно, что корректной является команда `n[0]=n1`, которой в качестве значения первому элементу `n[0]` массива указателей `n` присваивается ссылка на массив `n1`. Аналогично следует интерпретировать и команды `n[1]=n2` и `n[2]=n3`. Таким образом, элементами массива целочисленных указателей `n` являются ссылки на целочисленные массивы разного размера.

Теперь осталось выяснить смысл инструкции `n[1][2]`. Для этого достаточно обратить внимание на то обстоятельство, что инструкция `n[1]` есть ни что иное, как указатель на массив `n2` – в известном смысле это синонимы. Инструкция `n1[2]` означает третий элемент массива `n1`. Нетрудно догадаться, что `n[1][2]` – то же самое.

Примеры решения задач

Я не самонадеян, упаси бог. Просто я верю в то, что любая проблема решается, если все будут делать то, что я говорю.

Р. Рейган

Здесь рассматриваются задачи, в которых широко применяется концепция массивов для решения прикладных вопросов. В основном внимание уделено задачам линейной алгебры, и, в частности, действиям с матрицами и векторами, поскольку эти математические объекты как нельзя лучше описываются с помощью массивов. Задачи, подразумевающие использование массивов и указателей, также рассматриваются в следующих главах.

■ Скалярное произведение векторов

В листинге 3.11 приведен пример программы, которой вычисляется скалярное произведение двух векторов. Каждый вектор реализуется как одномерный массив из трех элементов. Элементы векторов вводятся пользователем. В результате выполнения программы отображается значение скалярного произведения этих векторов.

Листинг 3.11. Скалярное произведение векторов

```
#include <iostream>
using namespace std;
int main(){
    //Индексная переменная:
    int i;
    //Первый массив:
    double a[3];
    //Второй массив:
    double b[3];
    //Переменная для записи результата:
    double res=0;
    //Ввод элементов первого массива:
    cout<<"a = ";
    for(i=0;i<3;i++) cin>>a[i];
    //Ввод элементов второго массива:
    cout<<"b = ";
    for(i=0;i<3;i++) cin>>b[i];
    //Вычисление скалярного произведения:
    for(i=0;i<3;i++) res+=a[i]*b[i];
    //Отображение результата:
    cout<<"a.b = "<<res<<endl;
    return 0;
}
```

Результат выполнения программы может выглядеть следующим образом (жирным шрифтом выделен ввод пользователя):

```
a = 1 2 3
b = 4 5 6
a.b = 32
```

Напомним, что если векторы $\vec{a} = (a_1, a_2, a_3)$ и $\vec{b} = (b_1, b_2, b_3)$, то их скалярное произведение равно $\vec{a} \cdot \vec{b} = \sum_{n=1}^3 a_n b_n$, что и было использовано выше в программном коде. Сама программа достаточно проста: сначала

вводятся элементы двух векторов (для каждого вектора элементы вводятся в одну строку через пробел), после чего с помощью оператора цикла вычисляется скалярное произведение.

■ Векторное произведение

Результатом векторного произведения векторов $\vec{a} = (a_1, a_2, a_3)$ и $\vec{b} = (b_1, b_2, b_3)$ (обозначается $\vec{a} \times \vec{b}$ или $[\vec{a} \cdot \vec{b}]$) является вектор

$$\vec{c} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \vec{i}(a_2b_3 - a_3b_2) + \vec{j}(a_3b_1 - a_1b_3) + \vec{k}(a_1b_2 - a_2b_1), \text{ где}$$

через \vec{i} , \vec{j} и \vec{k} обозначены единичные орты декартовой системы координат. Программа, в которой вычисляется векторное произведение векторов, представлена в листинге 3.12.

Листинг 3.12. Векторное произведение

```
#include <iostream>
using namespace std;
int main(){
    //Индексная переменная:
    int i;
    //Первый массив:
    double a[3];
    //Второй массив:
    double b[3];
    //Третий массив (результат):
    double c[3];
    //Ввод элементов первого массива:
    cout<<"a = ";
    for(i=0;i<3;i++) cin>>a[i];
    //Ввод элементов второго массива:
    cout<<"b = ";
    for(i=0;i<3;i++) cin>>b[i];
    //Вычисление результата:
    for(i=0;i<3;i++) c[i]=a[(i+1)%3]*b[(i+2)%3]-a[(i+2)%3]*b[(i+1)%3];
    //Отображение результата:
    cout<<"[a.b] =";
    for(i=0;i<3;i++)
        cout<<" "<<c[i];
    cout<<endl;
    return 0;
}
```

По сравнению с предыдущим примером код изменился незначительно, однако есть ряд принципиальных новшеств. Во-первых, результатом является массив (отождествляемый с вектором). Во-вторых, обращаем внимание читателя на команду `c[i]=a[(i+1)%3]*b[(i+2)%3]-a[(i+2)%3]*b[(i+1)%3]`, которой в рамках оператора цикла определяются элементы вектора-результата. При выполнении программы можем получить, например, следующее:

```
a = 0 1 0
b = 1 0 0
[a.b] = 0 0 -1
```

Здесь, как и ранее, жирным шрифтом выделены значения, вводимые пользователем.

■ Сортировка массива методом пузырька

В качестве иллюстрации методов работы с одномерными массивами рассмотрим задачу о сортировке элементов массива. Существует несколько алгоритмов такой сортировки. В данном случае используем метод пузырька. Метод состоит в следующем. Каждый элемент массива сравнивается с соседним, и если первый из сравниваемых элементов больше второго, эти элементы меняются местами. После однократного перебора всех элементов самый большой элемент оказывается последним в массиве. Еще раз перебрав элементы массива (последний можно не трогать – он и так самый большой), на предпоследнее место перемещаем второй по величине элемент и так далее. Продолжая эту процедуру необходимое количество раз, добиваемся ситуации, когда элементы массива размещены в порядке возрастания. Данный алгоритм реализован в программе, код которой приведен в листинге 3.13.

Листинг 3.13. Сортировка массива

```
#include<iostream>
using namespace std;
int main(){
    const int m=10;
    int MyArray[m];
    int i,j,s;
    cout<<"Before:\n";
    // Исходный массив:
    for(i=0;i<m;i++){
        MyArray[i]=rand() % 20;
        cout<<MyArray[i]<<" ";}
```

```
// Сортировка массива:
for (j=1; j<= (m-1) ; j++)
for (i=0; i<m-j; i++)
if (MyArray[i]>MyArray[i+1]) {
    s=MyArray[i+1];
    MyArray[i+1]=MyArray[i];
    MyArray[i]=s; }
cout<<"\nAfter:\n";
// Массив после сортировки:
for (i=0; i<m; i++)
    cout<<MyArray[i]<<" ";
cout<<"\n";
return 0;
}
```

В начале программы инициализируется числовая константа, которая определяет размер массива. Далее случайными числами в диапазоне от 0 до 19 включительно заполняется массив. Одновременно с этим соответствующие элементы выводятся на экран для того, чтобы исходный массив можно было сравнить с отсортированным массивом.

Сортировка массива реализуется через два вложенных цикла. Внешний цикл `for (j=1; j<= (m-1) ; j++)` обеспечивает необходимое количество циклов перебора элементов массива. Через внутренний цикл `for (i=0; i<m-j; i++)` реализуется непосредственно перебор элементов массива. Причем верхняя граница для индекса элементов, которые определяются переменной `i`, зависит от количества переборов элементов массива, то есть от индексной переменной внешнего цикла.

В рамках двойного оператора цикла выполняется условный оператор, в котором сравниваются смежные элементы массива. Если значение элемента с меньшим индексом превышает следующий за ним элемент, они меняются местами. После сортировки массив выводится на экран. Результат выполнения приведенной программы может иметь следующий вид:

```
Before:
1 7 14 0 9 4 18 18 2 4
After:
0 1 2 4 4 7 9 14 18 18
```

Как правило, метод пузырька используют при сортировке не очень больших по размеру массивов. Данный метод, на самом деле, является не очень продуктивным с точки зрения времени выполнения программы, поэтому при работе с большими массивами лучше использовать иные подходы. В частности, в C++ для сортировки массива имеется специальная функция `qsort()`.

■ Умножение квадратных матриц

В общем случае произведением матрицы A размера $n \times m$ с элементами a_{ij} ($i = 1, 2, \dots, n, j = 1, 2, \dots, m$) и матрицы B размера $m \times p$ с элементами b_{jk} ($j = 1, 2, \dots, m, k = 1, 2, \dots, p$) является матрица C размера $n \times p$ с элементами $c_{ik} = \sum_{j=1}^m a_{ij}b_{jk}$ ($i = 1, 2, \dots, n, k = 1, 2, \dots, p$). В листинге 3.4 приведен пример программы, в которой вычисляется произведение двух квадратных матриц.

Листинг 3.14. Умножение квадратных матриц

```
#include <iostream>
using namespace std;
int main(){
    //Размер матриц:
    const int N=3;
    //Индексные переменные:
    int i,j,k;
    //Первая матрица:
    double A[N][N];
    //Вторая матрица:
    double B[N][N];
    //Третья матрица (результат):
    double C[N][N];
    //Ввод (построчный) элементов первой матрицы:
    cout<<"Matrix A:\n";
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            cin>>A[i][j];
        }
    }
    //Ввод (построчный) элементов второй матрицы:
    cout<<"Matrix B:\n";
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            cin>>B[i][j];
        }
    }
    //Вычисление произведения матриц:
    cout<<"Matrix C=AB:\n";
    for(i=0;i<N;i++){
        for(j=0;j<N;j++){
            C[i][j]=0;
            for(k=0;k<N;k++){
                C[i][j]+=A[i][k]*B[k][j];
            }
            //Вывод значения элемента на экран:
            cout<<C[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

В результате получаем следующее:

```
Matrix A:
1 1 1
1 0 1
0 0 1
Matrix B:
2 1 0
0 2 1
1 1 1
Matrix C=AB:
3 4 2
3 2 1
1 1 1
```

Жирным шрифтом выделены вводимые пользователем значения для элементов перемножаемых матриц.

■ Определитель матрицы

Определителем (детерминантом) квадратной матрицы $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$

размерами 2×2 называется число $\det(A) = a_{11}a_{22} - a_{12}a_{21}$, т.е. разница произведений диагональных и недиагональных элементов. В листинге 3.15 приведен код программы, с помощью которой вычисляется определитель квадратной матрицы.

Листинг 3.15. Определитель матрицы

```
#include <iostream>
using namespace std;
int main() {
    //Матрица:
    double A[2][2];
    //Определитель матрицы:
    double det;
    //Индексные переменные:
    int i, j;
    //Ввод (построчный) элементов матрицы:
    cout<<"Matrix A:\n";
    for(i=0; i<2; i++)
        for(j=0; j<2; j++)
            cin>>A[i][j];
    //Вычисление определителя матрицы:
    det=A[0][0]*A[1][1]-A[0][1]*A[1][0];
```

```
//Вывод значения определителя на экран:
cout<<"det (A) = "<<det<<endl;
return 0;
}
```

Пример выполнения программы может выглядеть так (жирным шрифтом выделены вводимые пользователем значения):

```
Matrix A:
1 2
3 4
det (A) = -2
```

Обычно задачи по вычислению определителя являются вспомогательными и решаются, например, при вычислении собственных чисел и собственных векторов матрицы.

■ Математическое ожидание для дискретной случайной величины

Дискретная случайная величина – это такая случайная величина, множество возможных значений которой является перечислимым (но не обязательно конечным). О случайной величине мы знаем, какие значения она в принципе может принимать и с какой вероятностью (это называется законом распределения случайной величины). Это важная информация, но она не позволяет однозначно установить, какое именно значение примет случайная величина.

Рассмотрим случайную величину ξ , которая может принимать значения $\xi_1, \xi_2, \dots, \xi_n$ соответственно с вероятностями p_1, p_2, \dots, p_n (для вероятностей имеет место соотношение $\sum_{i=1}^n p_i = 1$). Математическим ожиданием такой случайной величины называется число $M\xi = \sum_{i=1}^n \xi_i p_i$. Математиче-

ское ожидание очень часто отождествляют со средним значением случайной величины (если очень много раз провести опыт, в котором реализуется случайная величина, то среднее значение будет близко к математическому ожиданию). В листинге 3.16 приведен пример программы, с помощью которой для дискретной случайной величины на основе закона распределения вычисляется математическое ожидание этой случайной величины.

Листинг 3.16. Математическое ожидание

```
#include <iostream>
using namespace std;
int main() {
```

```

//Число реализуемых значений случайной величины:
const int n=5;
//Массив значений, массив вероятностей и
//математическое ожидание:
double xi[n],p[n],Mxi=0;
//Индексная переменная:
int i;
//Ввод реализуемых значений случайной величины:
cout<<"xi: ";
for(i=0;i<n;i++){
cin>>xi[i];
//Ввод вероятностей и вычисление
//математического ожидания:
cout<<"p: ";
for(i=0;i<n;i++){
    cin>>p[i];
    Mxi+=p[i]*xi[i];}
//Вывод результата:
cout<<"Mxi = "<<Mxi<<endl;
return 0;
}

```

Например, для случайной величины, которая принимает значения 0.5, 1.2, 2.5, 3.1 и 4.9 с вероятностями 0.1, 0.2, 0.1, 0.25 и 0.35 соответственно математическое ожидание равно 3.03:

```

xi: 0.5 1.2 2.5 3.1 4.9
p: 0.1 0.2 0.1 0.25 0.35
Mxi = 3.03

```

Важным является условие равенства единице суммы всех вероятностей реализации значений случайной величины.

■ Метод Ньютона

Существует метод решения алгебраических уравнений, который называется методом Ньютона (или итерационный метод Ньютона-Рафсона). Предположим, необходимо найти корень уравнения $f(x) = 0$ в окрестности точки $x = x_0$. Последовательный поиск приближений для корня уравнения выполняется по следующей схеме: в точке текущего приближения для корня уравнения проводится касательная к графику функции $f(x)$, точка пересечения этой касательной с осью абсцисс является новым приближением для корня. Такая последовательность действий сводится к рекуррентной

формуле $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$, где $f'(x)$ означает производную от функции

$f(x)$ по аргументу, x_n – приближение для корня уравнения на n -м итерационном шаге.

При решении уравнений методом Ньютона возникает некоторое неудобство, связанное с необходимостью вычислять (или задавать) кроме непосредственно функции $f(x)$ еще и производную $f'(x)$. В общем случае эта проблема решается разными способами. Здесь рассмотрим частную ситуацию, когда функция $f(x)$ является полиномом степени N относительно аргумента x , т.е. имеет место представление $f(x) = a_0 + a_1x + a_2x^2 + \dots + a_Nx^N = \sum_{k=0}^N a_kx^k$.

Поэтому для того, чтобы задать однозначно функцию $f(x)$ достаточно определить коэффициенты a_0, a_1, \dots, a_N , которые разумно реализовать в виде массива. Более того, если функция $f(x)$ является полиномом и коэффициенты этого полинома известны, легко найти и производную: производная $f'(x)$ также является полиномом, имеет место представление

$$f'(x) = a_1 + 2a_2x + 3a_3x^2 + \dots + Na_Nx^{N-1} = \sum_{k=0}^{N-1} (k+1)a_{k+1}x^k \equiv \sum_{k=0}^{N-1} b_kx^k,$$

причем коэффициенты полинома-производной b_k определяются на основании коэффициентов a_k функции-полинома через соотношение $b_k = (k+1)a_{k+1}$, $k = 0, 1, 2, \dots, N-1$. Именно эти зависимости использовались в программном коде, представленном в листинге 3.17. В данной программе реализован метод Ньютона для решения уравнения $f(x) = 0$ с функцией-полиномом $f(x)$. Коэффициенты полинома и начальное приближение вводятся пользователем.

Листинг 3.17. Решение уравнения методом Ньютона

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    //Порядок полинома:
    const int N=3;
    //Индексные переменные и число итераций:
    int i,k,n=10;
    //Аргумент, функция и производная:
    double x,f,df;
    //Массив коэффициентов полинома-функции:
    double a[N+1];
    //Массив коэффициентов полинома-производной:
    double b[N];
    //Ввод коэффициентов функции-полинома:
    cout<<"Function: ";
    cin>>a[0];
```

```

for(i=1;i<N+1;i++){
    cin>>a[i];
    //Вычисление коэффициентов для производной:
    b[i-1]=i*a[i];}
//Начальное приближение:
cout<<"Enter x0 = ";
cin>>x;
//Последовательные итерации:
for(k=1;k<=n;k++){
    f=a[0];
    df=0;
    for(i=1;i<N+1;i++){
        f+=a[i]*pow(x,i);
        df+=b[i-1]*pow(x,i-1);
    }
    x-=f/df;
}
//Результат:
cout<<"x = "<<x<<endl;
cout<<"f("<<x<<") = "<<f<<endl;
return 0;
}

```

Для проверки работы программы найдем один из корней полинома $f(x) = x^3 - 10x^2 + 27x - 18 = (x - 1)(x - 3)(x - 6)$. Вводим коэффициенты полинома и начальное приближение $x_0 = 0$, в результате чего находим корень $x = 1$:

```

Function: -18 27 -10 1
Enter x0 = 0
x = 1
f(1) = -4.21885e-015

```

При решении уравнений методом Ньютона многое зависит от того, насколько удачно выбрано начальное приближение для корня. Теоретически возможна ситуация, когда в соответствующей точке производная функции обращается в нуль (например, это имеет место в точках экстремума функции), и в этом случае, очевидно, возникает ошибка деления на ноль. Такие ситуации необходимо отслеживать.

■ Линейная регрессионная модель

При анализе статистических данных нередко приходится рассчитывать различные регрессионные модели. В общем случае при построении регрессионной модели для имеющегося набора значений аргумента $\{x_i\}$ ($i = 1, 2, \dots, n$) некоторой (как правило, неизвестной) функции и значений этой функции $\{y_i\}$ в указанных точках предлагается аналити-

ческая зависимость с неизвестными числовыми параметрами, которые необходимо рассчитать на основе статистических данных из тех соображений, чтобы данная аналитическая зависимость наилучшим образом описывала эти статистические данные. Для большей конкретности рассмотрим аналитическую функцию вида $f(x) = a\varphi(x) + b\psi(x)$, где функции $\varphi(x)$ и $\psi(x)$ известны и однозначно определены, а параметры a и b необходимо рассчитать на основе наборов значений $\{x_i\}$ и $\{y_i\}$ ($i = 1, 2, \dots, n$).

Для определения параметров регрессионной модели необходим критерий, который бы позволил сделать вывод, какая модель лучше, а какая хуже. Обычно используется метод наименьших квадратов. Суть метода сводится к тому, что параметры модели выбираются так, чтобы сумма квадратов отклонений значений в узловых точках y_i от значений в этих точках регрессионной зависимости $f(x_i)$ была минимальной. В данном случае имеем

$$\sum_{i=1}^n (f(x_i) - y_i)^2 \rightarrow \min \text{ или } \sum_{i=1}^n (a\varphi(x_i) + b\psi(x_i) - y_i)^2 \rightarrow \min. \text{ После}$$

минимизации по параметрам a и b получаем соотношения для этих пара-

$$\text{метров: } a = \frac{\sum_{i=1}^n y_i \psi_i \sum_{i=1}^n \varphi_i \psi_i - \sum_{i=1}^n y_i \varphi_i \sum_{i=1}^n \psi_i^2}{\left(\sum_{i=1}^n \varphi_i \psi_i\right)^2 - \sum_{i=1}^n \varphi_i^2 \sum_{i=1}^n \psi_i^2} \text{ и } b = \frac{\sum_{i=1}^n y_i \varphi_i - a \sum_{i=1}^n \varphi_i^2}{\sum_{i=1}^n \varphi_i \psi_i}.$$

Здесь для краткости обозначено $\varphi(x_i) \equiv \varphi_i$ и $\psi(x_i) \equiv \psi_i$. В частности, для линейной регрессионной модели $f(x) = ax + b$ получаем

$$a = \frac{\left(\frac{1}{n} \sum_{i=1}^n y_i\right) \left(\frac{1}{n} \sum_{i=1}^n x_i\right) - \frac{1}{n} \sum_{i=1}^n y_i x_i}{\left(\frac{1}{n} \sum_{i=1}^n x_i\right)^2 - \frac{1}{n} \sum_{i=1}^n x_i^2} \text{ и } b = \frac{\frac{1}{n} \sum_{i=1}^n y_i x_i - a \frac{1}{n} \sum_{i=1}^n x_i^2}{\frac{1}{n} \sum_{i=1}^n x_i}.$$

Обращаем внимание читателя, что $\frac{1}{n} \sum_{i=1}^n x_i$ представляет собой среднее значение по массиву $\{x_i\}$, $\frac{1}{n} \sum_{i=1}^n y_i$ — среднее значение по массиву $\{y_i\}$, $\frac{1}{n} \sum_{i=1}^n x_i^2$ — среднее значение квадратов по массиву $\{x_i\}$, $\frac{1}{n} \sum_{i=1}^n y_i x_i$ — среднее значение по массиву произведений узловых точек x_i и статистических значений функции в этих точках y_i . В листинге 3.18 приведен пример программы, в которой на основе набора данных вычисляются параметры линейной регрессионной модели.

Листинг 3.18. Линейная регрессионная модель

```

#include <iostream>
using namespace std;
int main(){
    //Размер массива:
    const int n=10;
    //Индексная переменная:
    int i;
    //Массивы статистических значений:
    double x[n],y[n];
    //Параметры модели:
    double a,b;
    //Средние значения:
    double Sx=0,Sy=0,Sxy=0,Sxx=0;
    //Ввод статистических данных:
    cout<<"x = ";
    for(i=0;i<n;i++){
        cin>>x[i];
        cout<<"y = ";
        for(i=0;i<n;i++){
            cin>>y[i];
            //Вычисление параметров модели:
            for(i=0;i<n;i++){
                Sx+=x[i];
                Sy+=y[i];
                Sxy+=x[i]*y[i];
                Sxx+=x[i]*x[i];
            }
        }
    }
    Sx/=n;
    Sy/=n;
    Sxy/=n;
    Sxx/=n;
    a=(Sx*Sy-Sxy)/(Sx*Sx-Sxx);
    b=(Sxy-a*Sxx)/Sx;
    //Результат:
    cout<<"a = "<<a<<endl;
    cout<<"b = "<<b<<endl;
    return 0;
}

```

Регрессионная модель тем точнее описывает данные, чем ближе реальная статистическая зависимость между параметрами x и y к той функциональной зависимости, что использовалась при построении модели. Другими словами, чтобы статистические данные хорошо ложились на аппроксимирующую кривую, необходимо удачно подобрать вид этой кривой. К сожалению, точных рецептов для этого не существует.

Чтобы проверить работу программы, вводим значения узловых точек и функции в этих точках такие, что соответствуют зависимости $y = 2x + 1$. В этом случае программа должна выдать значения $a = 2$ и $b = 1$. Результат имеет следующий вид (жирным шрифтом выделен ввод пользователя):

```
x = -4 -3 -2 -1 0 1 2 3 4 5
y = -7 -5 -3 -1 1 3 5 7 9 11
a = 2
b = 1
```

Как видим, совпадение полное. На практике статистические данные содержат случайные составляющие, что существенно нарушает идиллию.

Резюме

*Уже светает, пора подводить
неутешительные итоги.*

Из к/ф «Гараж»

1. Указателем называется переменная, значением которой является адрес ячейки памяти. С помощью указателя можно обращаться к содержимому ячейки, адрес которой является значением указателя.
2. При объявлении указателя указывается тип значения, хранимого в ячейке, на которую ссылается указатель, а перед именем соответствующей переменной-указателя размещается оператор *. Для получения доступа к значению, записанному в ячейку, на которую ссылается указатель, перед именем указателя также указывают оператор *. Адрес ячейки, в которую записана переменная, можно получить, указав перед именем этой переменной оператор &.
3. С переменными-указателями можно выполнять простые арифметические действия, результат которых определяется правилами адресной арифметики.
4. Указатель может в качестве значения содержать адрес переменной-указателя. В этом случае говорят о многоуровневой адресации. Объявление указателя на указатель выполняется с использованием двух операторов * перед именем переменной.
5. Ссылка является альтернативным способом обращения к переменным. Объявление ссылки выполняется одновременно с ее инициализацией. Перед именем ссылки указывается оператор &, значением ссылки является имя переменной, на которую выполняется ссылка. После выполнения ссылки на переменную к переменной можно обращаться через два имени: непосредственно имя переменной и имя переменной-ссылки.

6. Массивом называется набор однотипных элементов, обращение к которым выполняется через одно общее имя (имя массива) с указанием индекса элемента в массиве.
7. При объявлении массива указывается тип элементов, имя массива и размера массива (размер указывается после имени в квадратных скобках). Для многомерных массивов размер указывается для каждой размерности.
8. Размер массива должен быть известен на момент компиляции (такие массивы называются статическими).
9. Обращение к элементу массива выполняется через имя массива с указанием индекса элемента (в квадратных скобках). Если индексов несколько, для каждого из них используется отдельная пара квадратных скобок.
10. Индексация элементов всегда начинается с нуля.
11. Все элементы массива в памяти размещаются последовательно друг за другом.
12. Проверка на предмет выхода за пределы массива не выполняется.
13. Имя массива является указателем на его первый элемент (первый элемент имеет индекс 0).
14. Указатели можно индексировать. Это обстоятельство часто используется при работе с массивами.
15. Массивы при объявлении можно инициализировать: значения элементов массива указываются в виде списка, заключенного в фигурные скобки.
16. В C++ массив символов является особым видом массива, поскольку используется для реализации текстовых строк. Такой символьный массив заканчивается нуль-символом `'\0'` (не путать с нулем!), который является признаком окончания строки.

Контрольные вопросы

С болью в сердце зачитываю список.

Из к/ф «Гараж»

1. Что такое указатель? Как указатель объявляется и как используется?
2. Какие операции допустимы с указателями?

3. Что такое многоуровневая адресация?
4. Что такое ссылка? Чем она отличается от указателя?
5. Что такое массив? Какие бывают массивы?
6. В чем особенность массивов в C++? Выполняется ли в C++ проверка выхода за пределы массива?
7. Как индексируются элементы массива?
8. Как объявляется одномерный массив? Как выполняется обращение к элементам массива?
9. Как объявляются двумерные массивы? Как выполняется обращение к элементам двумерного массива?
10. Как выполняется инициализация массивов?
11. В чем особенность имени массива? Как имя массива связано с указателем на первый его элемент?
12. В чем особенность массивов, состоящих из символов?
13. В чем особенность массивов, состоящих из указателей?

Задачи для самостоятельного решения

Задача 1. Написать программу для вычисления модуля вектора. Компоненты вектора вводятся пользователем. Напомним, что модулем вектора $\vec{a} = (a_1, a_2, a_3)$ называется число $|\vec{a}| = \sqrt{a_1^2 + a_2^2 + a_3^2}$.

Задача 2. Написать программу для вычисления угла φ между векторами $\vec{a} = (a_1, a_2, a_3)$ и $\vec{b} = (b_1, b_2, b_3)$. Компоненты векторов вводятся пользователем. При вычислениях воспользоваться соотношением $\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cos(\varphi)$, где через $\vec{a} \cdot \vec{b}$ обозначено скалярное произведение векторов.

Задача 3. Написать программу для выполнения транспонирования матрицы. При транспонировании матрицы A с элементами a_{ij} выполняется замена вида $a_{ij} \rightarrow a_{ji}$, т.е. матрица «переворачивается» относительно главной диагонали.

Задача 4. Написать программу для вычисления собственных λ чисел квадратной матрицы $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ размерами 2×2 . Собственные числа

ищутся из условия $\det(A - \lambda E) = 0$, где $E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ единичная матрица.

Условие $\det(A - \lambda E) = 0$ эквивалентно в случае матрицы размеров 2×2 квадратному уравнению $(a_{11} - \lambda)(a_{22} - \lambda) = a_{12}a_{21}$. Предусмотреть ситуацию, когда уравнение решение не имеет (решения комплексные).

Задача 5. Написать программу для вычисления определителя матрицы размерами 3×3 . Воспользоваться тем, что

для матрицы вида $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$ определитель равен

$$\det(A) = a_{11} \det \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - a_{12} \det \begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} + a_{13} \det \begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}.$$

Задача 6. Написать программу для поиска наибольшего элемента массива. Массив заполнить случайными числами.

Задача 7. Написать программу для поиска наименьшего элемента массива. Массив заполнить случайными числами.

Задача 8. Написать программы для вычисления дисперсии случайной величины, по заданному закону распределения случайной величины. Напомним, что если случайная величина ξ принимает значения $\xi_1, \xi_2, \dots, \xi_n$ с вероятностями p_1, p_2, \dots, p_n , то дисперсией такой случайной величины является

число $D\xi = M(\xi - M\xi)^2 = M\xi^2 - (M\xi)^2$, где $M\xi = \sum_{i=1}^n \xi_i p_i$ — математическое

ожидание случайной величины ξ , $M\xi^2 = \sum_{i=1}^n \xi_i^2 p_i$ — математическое

ожидание квадрата этой случайной величины. Закон распределения случайной величины вводится пользователем (массив значений и массив

вероятностей). При этом должно выполняться условие $\sum_{i=1}^n p_i = 1$.

Задача 9. Моментом порядка k случайной величины ξ называется число $M\xi^k = \sum_{i=1}^n \xi_i^k p_i$, где случайная величина ξ принимает значения $\xi_1, \xi_2, \dots, \xi_n$ с вероятностями p_1, p_2, \dots, p_n соответственно. Написать программу для вычисления момента порядка k для случайной величины с заданным законом распределения. Порядок момента и закон распределения вводятся пользователем.

Задача 10. Центральным моментом порядка k случайной величины ξ называется число $M(\xi - M\xi)^k = \sum_{i=1}^n (\xi_i - M\xi)^k p_i$, где случайная величина ξ принимает значения $\xi_1, \xi_2, \dots, \xi_n$ с вероятностями p_1, p_2, \dots, p_n соответственно, а $M\xi = \sum_{i=1}^n \xi_i p_i$ – математическое ожидание этой случайной величины. Написать программу для вычисления центрального момента порядка k для случайной величины с заданным законом распределения. Порядок момента и закон распределения вводятся пользователем.

Задача 11. Написать программу для вычисления среднего арифметического для элементов массива. Напомним, что среднее значение определяется как $\langle x \rangle = \frac{1}{n} \sum_{i=1}^n x_i$, где x_i – элементы массива, $i = 1, 2, \dots, n$, n – количество элементов массива.

Задача 12. Написать программу для вычисления среднеквадратичного значения для элементов массива. Напомним, что среднеквадратичное значение определяется как $\sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2}$, где x_i – элементы массива, $i = 1, 2, \dots, n$, n – количество элементов массива.

Задача 13. Написать программу для вычисления выборочной дисперсии m_2 , которая вычисляется по формуле $m_2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$, где через $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$ обозначено выборочное среднее, x_i – элементы массива (выборки), $i = 1, 2, \dots, n$, n – количество элементов массива (выборки).

Задача 14. Написать программу для вычисления значения полинома. Пользователем вводятся коэффициенты массива и аргумент, для которого вычисляется значение полинома. Предусмотреть возможность многократного вычисления значения (например, после вычисления значения полинома выводится запрос на продолжение работы).

Задача 15. Написать программу для вычисления значения производной от полинома. Пользователем вводятся коэффициенты массива и аргумент, для которого вычисляется значение производной полинома. Предусмотреть возможность многократного вычисления значения (например, после вычисления значения производной полинома выводится запрос на продолжение работы).

Задача 16. Написать программу для решения методом Ньютона уравнения $f(x) = 0$, где функция $f(x)$ имеет вид $f(x) = \sum_{k=0}^N a_k x^{2k}$. Коэффициенты полинома и начальная точка вводятся пользователем.

Задача 17. Написать программу для решения методом Ньютона уравнения $f(x) = 0$, где функция $f(x)$ имеет вид $f(x) = \sum_{k=-1}^N a_k x^{2k+1}$. Коэффициенты полинома и начальная точка вводятся пользователем.

Задача 18. Написать программу для решения методом Ньютона уравнения $f(x) = 0$, где функция $f(x)$ имеет вид $f(x) = \sum_{k=-1}^N a_k x^k$. Коэффициенты полинома и начальная точка вводятся пользователем.

Задача 19. Написать программу для вычисления параметров a и b регрессионной модели вида $f(x) = a + \frac{b}{x}$ (см. задачу про линейную регрессионную модель). Значения $\{x_i\}$ и $\{y_i\}$ вводятся пользователем.

Задача 20. Написать программу для вычисления параметров a и b регрессионной модели вида $f(x) = ax + \frac{b}{x}$ (см. задачу про линейную регрессионную модель). Значения $\{x_i\}$ и $\{y_i\}$ вводятся пользователем.

Глава 4

Функции

Использование функций при создании программ существенно упрощает структуру программного кода и позволяет решать ряд принципиальных задач. Само понятие функции в C++, как и в большинстве других языков программирования, достаточно точно соответствует математическому определению функции, хотя некоторые особенности, безусловно, существуют. Частично они связаны с тем, что это функции программные, некоторая специфика также связана с языком C++.

Под функцией подразумевают именованный программный код, который может многократно вызываться в программе. Функции реализуются отдельными программными блоками, могут иметь аргументы и возвращать значения в качестве результата (а могут и не возвращать, результатом функции может быть практически все, кроме массива).

Далее речь пойдет о том, как функции объявляются и используются.

Объявление и использование функций

Прибегните к совету Конька-Горбушка: возьмите книжку и прочитайте – там все очень доступно написано.

Из к/ф «Чародеи»

Формат объявления функции в C++ имеет такую структуру: указывается тип значения, которое возвращает функция, название функции, в круглых скобках список аргументов (с указанием типа аргументов). Программный код функции указывается в блоке из фигурных скобок:

```
тип_результата имя_функции(тип аргумент1, тип аргумент2, ...) {  
    код функции  
}
```

В листинге 4.1 приведен пример программы с объявлением функции `msum()`, которая возвращает в качестве результата значение суммы натуральных чисел. Количество слагаемых в сумме определяется целочисленным аргументом функции.

Листинг 4.1. Код программы с функцией для вычисления суммы натуральных чисел

```
#include<iostream>
using namespace std;
int msum(int n){
    int s=0;
    for(int i=1;i<=n;i++) s+=i;
    return s;}
int main(){
    int n;
    cout<<"Enter n = ";
    cin>>n;
    cout<<"Sum is "<<msum(n)<<"\n";
    return 0;
}
```

Основу кода функции составляет оператор цикла `for(int i=1;i<=n;i++) s+=i`. Индексная переменная `i` пробегает с единичным шагом дискретности значения от 1 до `n` (аргумент функции). На каждом итерационном шаге переменная `s`, инициализированная с нулевым начальным значением, увеличивается на `i` (команда `s+=i`). Значение именно этой переменной `s` возвращается в виде результата функции (команда `return s`). В основном методе `main()` программы созданная функция `msum()` вызывается в команде `cout<<"Sum is "<<msum(n)<<"\n"` для вычисления суммы натуральных чисел. Предварительно значение целочисленной переменной `n` вводится пользователем с клавиатуры.

Функция не всегда должна возвращать результат. Если функция результат не возвращает, в качестве типа функции указывается `void`. В этом случае функция напоминает процедуру. Пример такой функции приведен в листинге 4.2.

Листинг 4.2. Функция не возвращает результат

```
#include<iostream>
using namespace std;
void msum2(){
    int n;
    cout<<"Enter n = ";
    cin>>n;
    int s=0;
    for(int i=1;i<=n;i++) s+=i;
    cout<<"Sum is "<<s<<"\n";
}
int main(){
```



```
msum2 ();
return 0;
}
```

Более того, функция не только не возвращает результат – у нее еще и нет аргументов (тем не менее, скобки после названия функции все равно указываются, причем как при объявлении функции, так и при ее вызове).

Функциональность программы после внесения исправлений не изменилась: на экране выводится приглашения для пользователя ввести натуральное число (верхнюю границу для суммы натуральных чисел), это число считывается, после чего вычисляется сумма натуральных чисел. Что изменилось, так это организация программного кода. Фактически главный метод программы состоит всего из одной команды (если не считать стандартную инструкцию `return 0`) по вызову функции `msum2 ()`. Вся функциональность программы спрятана в этой функции. Вначале в рамках вызова функции `msum2 ()` запрашивается значение целочисленной переменной `n` (теперь это локальная переменная функции). Затем вводится переменная `s` для записи значения суммы натуральных чисел, выполняется оператор цикла и на экран выводится значение суммы (переменная `s`). На этом выполнение функции заканчивается.

Все переменные, использованные в теле функции, являются локальными. Это означает, что они доступны только в теле функции, но никак не за ее пределами. Вообще же доступность переменных определяется простым правилом: переменные доступны в пределах того блока, где они объявлены. Напомним, что блок в C++ ограничивается парой фигурных скобок.

Несмотря на то что функция, объявленная с типом `void`, результат не возвращает, она может содержать инструкцию `return`, и не одну. В этом случае инструкция `return` является командой завершения функции. Пример такой функции приведен в листинге 4.3.

Листинг 4.3. Функция с инструкцией `return` не возвращает результат

```
void InvFunc(double z){
    if(z==0){
        cout<<"Division by zero!"<<endl;
        return;}
    double x;
    x=1/z;
    cout<<"1/z ="<<x<<endl;
}
```

У функции `InvFunc ()` один аргумент типа `double`, а в качестве типа возвращаемого результата указано `void` (функция результат не возвращает).

В результате выполнения функции, в зависимости от значения аргумента, возможны два сценария. При ненулевом аргументе на экран выводится значение, обратное к аргументу. Если аргумент у функции нулевой, выводится сообщение соответствующего содержания.

Проверка аргумента на предмет равенства нулю осуществляется в условном операторе. Блок условного оператора, выполняющийся при нулевом аргументе, состоит из двух команд: выводится сообщение (команда `cout<<"Division by zero!"<<endl`) и затем завершается выполнение функции (команда `return`).

Если аргумент ненулевой, объявляется переменная `x` типа `double`. В качестве значения ей присваивается величина, обратная к аргументу функции, после чего полученное значение выводится на экран.

В обеих командах вывода в функции использована инструкция завершения строки `endl`. Кроме того, в данном случае фактически инструкция `return` использована как альтернатива к `else`-блоку условного оператора: можно было бы ту часть кода, что выполняется при ненулевом аргументе функции, разместить в `else`-блоке условного оператора, отказавшись при этом от инструкции завершения функции `return`. Тем не менее, в более сложных программах использование такой инструкции часто бывает не только оправданным, но и существенно упрощает структуру кода.

Что касается непосредственно объявления функции, то в C++ допускается в начале программы указывать прототип функции (прототипом функции называется «шапка» с типом возвращаемого результата, именем функции и списком аргументов), а непосредственно объявление функции переносить в конец программы. Корректным является такой программный код (листинг 4.4).

Листинг 4.4. Функция объявлена в конце программы

```
#include <iostream>
using namespace std;
//Прототип функции InvFunc():
void InvFunc(double z);
//Функция InvFunc() использована в программе:
int main(){
    double s;
    cout<<" Enter number: ";
    cin>>s;
    InvFunc(s);
    return 0;
}
//Объявление функции InvFunc():
```

```
void InvFunc(double z){
    if(z==0){
        cout<<"Division by zero!"<<endl;
        return;}
    double x;
    x=1/z;
    cout<<"1/z ="<<x<<endl;
}
```

По большому счету, описывать функцию можно где угодно – главное, чтобы ее прототип был указан до ее первого использования.

Если описание функции и ее прототип разнесены в программе, в прототипе при перечислении аргументов их формальные названия указывать не обязательно – достаточно указать только тип аргумента. Например, вместо прототипа `void InvFunc(double z)` в листинге 6.4 можно было указать прототип `void InvFunc(double)`. Так же поступают в случае, когда у функции несколько аргументов: в списке аргументов перечисляют через запятую только их типы, без названий.

Механизмы передачи аргументов

В C++ существует два механизма передачи аргументов функциям: по значению и через ссылку. При передаче аргумента функции по значению при вызове функции для переменных, которые указаны ее аргументами, создаются копии, которые фактически и передаются функции. После завершения выполнения кода функции эти копии уничтожаются (выгружаются из памяти). При передаче аргументов функции по ссылке функция получает непосредственный доступ (через ссылку) к переменным, указанным аргументами функции. **С практической точки зрения разница между этими механизмами заключается в том, что при передаче аргументов по значению изменить передаваемые функции аргументы в теле самой функции нельзя, а при передаче аргументов по ссылке – можно. По умолчанию используется механизм передачи аргументов функции по значению.**

Проиллюстрируем сказанное на конкретном примере. Рассмотрим программный код, приведенный в листинге 4.5.

Листинг 4.5. Передача аргумента по значению

```
#include <iostream>
using namespace std;
//Аргумент передается по значению:
int incr(int m){
```

```

    m=m+1;
    return m;
}
int main() {
    int n=5;
    cout<<"n ="<<incr(n)<<endl;
    cout<<"n ="<<n<<endl;
    return 0;
}

```

Программный код функции `incr()` предельно прост: функция в качестве значения возвращает целочисленную величину, на единицу превышающую аргумент функции. Особенность программного кода состоит в том, что в теле функции выполняется команда `m=m+1`, которой с формальной точки зрения аргумент функции `m` увеличивается на единицу, и именно это значение возвращается в качестве результата. Поэтому с точки зрения здравой логики после вызова функции переменная, переданная ей в качестве аргумента, должна увеличиться на единицу. Однако это не так. Вывод результатов выполнения программы на экран будет выглядеть так:

```

n =6
n =5

```

Если с первой строкой проблем не возникает, то вторая выглядит несколько неожиданно. Хотя это только на первый взгляд. Остановимся подробнее на том, что происходит при выполнении программы.

В начале главного метода `main()` инициализируется со значением 5 целочисленная переменная `n`. Далее на экран выводится результат вычисления выражения `incr(n)` и затем значение переменной `n`. Поскольку функция `incr()` возвращает значение, на единицу большее аргумента, результат этого выражения равен 6. Но почему же тогда не меняется переменная `n`? Все дело в механизме передачи аргумента функции. Поскольку аргумент передается по значению, при выполнении инструкции `incr(n)` для переменной `n` автоматически создается копия, которая и передается аргументом функции `incr()`. В соответствии с кодом функции значение переменной-копии увеличивается на единицу и полученное значение возвращается в качестве результата функции. Как только результат функцией возвращен, переменная-копия прекращает свое существование. Поэтому функция вычисляется корректно, а переменная-аргумент не меняется!

Для того чтобы аргумент передавался не по значению, а по ссылке, перед именем соответствующего аргумента необходимо указать оператор `&`. В листинге 4.6 приведен практически тот же программный код, что и в листинге 4.5, однако аргумент функции `incr()` в этом случае передается по ссылке.

Листинг 4.6. Передача аргумента по ссылке

```
#include <iostream>
using namespace std;
//Аргумент передается по ссылке:
int incr(int &m){
    m=m+1;
    return m;
}
int main(){
    int n=5;
    cout<<"n ="<<incr(n)<<endl;
    cout<<"n ="<<n<<endl;
    return 0;
}
```

Результат выполнения программы выглядит так:

```
n =6
n =6
```

Поскольку аргумент функции передается по ссылке, то все манипуляции в теле функции выполняются не с копией аргумента, а непосредственно с аргументом. Таким образом, вызов функции `incr(n)` не только возвращает в качестве результата увеличенное на единицу значение аргумента, но приводит к тому, что этот аргумент действительно увеличивается на единицу.

Если у функции несколько аргументов, часть из них (или все) могут передаваться по ссылке, а часть – по значению. Пример приведен в листинге 4.7.

Листинг 4.7. Разные механизмы передачи аргументов

```
#include <iostream>
using namespace std;
//Аргумент передается по ссылке и по значению:
void change(int &m, int n){
    int k;
    k=n;
    n=m;
    m=k;
    cout<<"m ="<<m<<endl;
    cout<<"n ="<<n<<endl;
}
int main(){
    int m=3,n=5;
    change(m,n);
    cout<<"m ="<<m<<endl;
    cout<<"n ="<<n<<endl;
    return 0;
}
```

В результате выполнения программы на экране появятся следующие строки:

```
m =5
n =3
m =5
n =5
```

У функции `change()` два целочисленных аргумента: первый передается по ссылке, а второй передается по значению. Действие функции состоит в том, что значения аргументов меняются местами: первому аргументу присваивается значение второго аргумента и наоборот. Но поскольку по ссылке передается только первый аргумент, то изменяется значение только этого аргумента – второй остается неизменным.

Обращаем внимание читателя на то, что наличие оператора `&` перед именем аргумента – всего лишь инструкция для определения механизма передачи аргумента. Это не влияет на способ обращения к аргументу или использования аргумента в теле функции. В этом смысле передача аргумента по ссылке отличается от передачи аргументом функции указателя на переменную. Об этом речь пойдет в следующем разделе.

Передача указателя аргументом функции

В качестве аргументов функции могут передаваться указатели. При передаче указателя аргументом функции перед именем указателя указывается оператор `*`. Пример передачи указателя аргументом функции приведен в листинге 4.8.

Листинг 4.8. Передача указателя аргументом функции

```
#include <iostream>
using namespace std;
//Аргументом является указатель:
int incr(int *m){
    *m=*m+1;
    return *m;
}
int main(){
    int n=5;
    cout<<"n ="<<incr(&n)<<endl;
    cout<<"n ="<<n<<endl;
    return 0;
}
```

Фактически представленный код является эквивалентом кода программы, представленной в листинге 4.6, однако вместо передачи аргумента по ссылке аргументом функции передается указатель. Прототип функции выглядит как `int incr(int *m)`. Теперь `m` является указателем на переменную целого типа. Чтобы получить доступ к значению этой переменной, используем инструкцию вида `*m`, что и было сделано в программе.

При вызове функции ее аргументом указывается не переменная `n`, а адрес этой переменной, который получаем с помощью инструкции `&n`.

Хочется сделать несколько замечаний относительно особенностей указателей как аргументов функции. Дело в том, что указатель, будучи переданным аргументом функции, передается, как обычные переменные, по значению. Убедиться в этом можно с помощью кода, представленного в листинге 4.9.

Листинг 4.9. Указатель передается аргументом функции по значению

```
#include <iostream>
using namespace std;
void test(int *n){
    cout<<"Address: "<<&n<<"\n";
}
int main(){
    int n=1;
    int *p;
    p=&n;
    test(p);
    cout<<"Address: "<<&p<<"\n";
    return 0;
}
```

Основу кода составляет функция `test()`, аргументом которой является целочисленный указатель (аргумент объявлен как `int *n`). В результате выполнения функции на экран выводится адрес аргумента функции (адрес получаем через инструкцию `&n`). Адрес аргумента – это указатель на указатель. Таким образом, при вызове функции на экране появляется сообщение с адресом переменной, которая реально обрабатывается функцией.

В главном методе программы с единичным значением инициализируется целочисленная переменная `n`. Кроме этого объявляется указатель `p` на целочисленную переменную и этой переменной в качестве значения присваивается адрес переменной `n`. Переменная-указатель `p` передается аргументом функции `test()` и, кроме этого, на экран выводится адрес переменной `p` (инструкция `&p`). Результат выполнения программы может иметь следующий вид:

```
Address: 0012FF28
```

```
Address: 0012FF78
```

Конкретные значения не важны, важно то, что они различны. Если бы в функцию передавалась не копия переменной-указателя, то в обоих случаях адреса бы совпадали. Таким образом, как и в случае с обычными переменными, указатель по умолчанию передается аргументом функции по значению. Тем не менее, он обеспечивает доступ из функции к исходной переменной, на которую ссылается, поскольку копия этого указателя ссылается на ту же самую переменную.

Наконец, чтобы передать аргумент-указатель по ссылке, а не по значению, можно в качестве прототипа функции указать `void test(int *&n)`.

Передача массива аргументом функции

Нередко в качестве аргументов функций указываются массивы. Учитывая то обстоятельство, что имя массива является ссылкой на первый его элемент, существует некоторая свобода в способе передачи массива аргументом функции. Хотя, если смотреть в корень проблемы, то все способы базируются на одном механизме. Тем не менее, рассмотрим все варианты. Первый и наиболее прямолинейный проиллюстрирован в листинге 4.10.

Листинг 4.10. Передача аргументом массива: явное указание размера

```
#include <iostream>
using namespace std;
//При объявлении массива явно указан размер:
void show(int n[5]){
    for(int i=0;i<5;i++)
        cout<<"n["<<i<<"]="<<n[i]<<endl;
}
int main(){
    int n[5]={1,2,3,4,5};
    show(n);
    return 0;
}
```

В программе объявляется функция `show()`, аргументом которой указан целочисленный массив из пяти элементов. Прототип функции имеет вид `void show(int n[5])`. Передаваемый массив указывается вместе с размером. В результате выполнения программы в столбик выводятся значения элементов массива:


```

n[0]=1
n[1]=2
n[2]=3
n[3]=4
n[4]=5

```

Ничего не изменится, если при объявлении функции размер в явном виде не указывать (листинг 4.11).

Листинг 4.11. Передача аргументом массива: размер не указан

```

#include <iostream>
using namespace std;
//При объявлении массива размер не указан:
void show(int n[]){
    for(int i=0;i<5;i++){
        cout<<"n["<<i<<"]="<<n[i]<<endl;
    }
}
int main(){
    int n[5]={1,2,3,4,5};
    show(n);
    return 0;
}

```

Причина кроется в том, что при передаче массива аргументом и в том, и в другом случае на самом деле передается ссылка на первый элемент массива. Поэтому особого значения не имеет, указан размер массива или нет – в C++ все равно проверки на предмет выхода за пределы массива нет. Поэтому разумнее, принимая во внимание сказанное, передавать массив в виде указателя, благо таковым является имя массива. Пример соответствующего кода приведен в листинге 4.12.

Листинг 4.12. Передача аргументом массива в виде указателя

```

#include <iostream>
using namespace std;
//Аргументом указано имя массива и размер:
void show(int *n,int m){
    for(int i=0;i<m;i++){
        cout<<"n["<<i<<"]="<<n[i]<<endl;
    }
}
int main(){
    int n[5]={1,2,3,4,5};
    show(n,5);
    show(n,3);
    return 0;
}

```

Прототип функции имеет вид `void show(int *n, int m)`. В соответствии с этим прототипом у функции два аргумента: целочисленный указатель (имя массива) и целое число (размер массива). В теле функции в операторе цикла верхняя граница изменения индексной переменной указана как `m`. В главном модуле функция `show()` вызывается дважды: первый раз в формате `show(n, 5)` и `show(n, 3)`. В результате получаем:

```
n[0]=1
n[1]=2
n[2]=3
n[3]=4
n[4]=5
n[0]=1
n[1]=2
n[2]=3
```

Здесь использовано свойство, что, во-первых, имя массива является указателем на первый его элемент, а во-вторых, указатели можно индексировать. Более того, изменяя второй аргумент, можем изменять количество обрабатываемых элементов массива (главное не выйти при этом за пределы массива).

Передача многомерных, и в частности двумерных, массивов аргументами функции осуществляется по следующему принципу: указываются все размеры массива, кроме первого. Такая явная индексация связана со способом интерпретации многомерных массивов в C++. Так, двумерный массив – это массив массивов. Поэтому, как и в случае одномерного массива, первый индекс является необязательным, поскольку функции на самом деле передается ссылка на первый элемент массива (для многомерных массивов этот первый элемент сам является массивом). Все последующие индексы нужны для того, чтобы компилятор мог корректно выделить место под элементы массива. Пример передачи двумерного массива аргументом функции приведен в листинге 4.13.

Листинг 4.13. Передача аргументом двумерного массива

```
#include <iostream>
using namespace std;
void show2(int n[][3], int size){
    int i, j;
    for(i=0; i<size; i++){
        for(j=0; j<3; j++){
            cout<<n[i][j]<<" ";
        }
        cout<<"\n";
    }
}
```

```
int main(){
    int n[][3]={ {1,2,3},
                  {4,5,6} };
    show2(n,2);
    return 0;
}
```

Функцией `show2()` выводится на экран двумерный массив. В результате на экране появится сообщение

```
1 2 3
4 5 6
```

Функции аргументом передается двумерный массив: инструкция `int n[][3]` в прототипе функции содержит явно указанный размер массива по второму индексу. Кроме двумерного массива, функции в качестве значения передается целочисленная переменная `size`, определяющая размер массива по первому индексу. Обращение к функции в главном методе осуществляется в формате `show2(n,2)`, где `n` – предварительно инициализированный двумерный массив.

Передача строки аргументом функции

Поскольку один из вариантов реализации текстовых строк подразумевает их представление в виде символьного массива, нетрудно догадаться, что текстовые строки могут передаваться аргументами функциям практически так же, как и прочие массивы. Хотя имеются и свои особенности. В основном они касаются способов обработки таких символьных массивов. Пример приведен в листинге 4.14.

Листинг 4.14. Передача аргументом символьного массива

```
#include <iostream>
using namespace std;
int length(char *str){
    for(int s=0;*str;s++,str++);
    return s;
}
int main(){
    char str[20]="This is a string";
    cout<<"Length is "<<length(str)<<endl;
    return 0;
}
```

Функцией `length()` для массива, указанного аргументом функции, подсчитывается количество символов. Для этого используется оператор цикла, в блоке инициализации которого с нулевым начальным значением инициализируется переменная `s`. Переменная необходима для подсчета количества символов. В качестве проверяемого условия указана инструкция `*str` (где `str` есть аргумент функции – указатель на данные типа `char`). В данном случае использовано то обстоятельство, что фактическая текстовая строка, представленная в виде символьного массива, заканчивается нуль-символом. Поэтому достижение конца строки эквивалентно тому, что значение соответствующего элемента массива равняется нулю. В блоке инкремента две команды: одной на единицу увеличивается значение переменной-счетчика `s`, а второй командой на единицу увеличивается значение указателя `str`. Напомним, что адресная арифметика имеет свои правила: увеличение указателя на единицу означает переход к следующей ячейке памяти. Поскольку элементы массива размещаются в памяти подряд один за другим, изменение указателя на единицу означает переход к следующему элементу массива. В качестве значения функции возвращается значение переменной `s`. Также обращаем внимание читателя на то обстоятельство, что размер массива аргументом функции не передается. В данном случае в этом нет необходимости в силу двух причин. Во-первых, сам по себе размер массива не очень актуален, поскольку речь идет о символьном массиве, в котором фактические данные занимают не все ячейки. Во-вторых, поскольку существует четкий признак окончания строки (имеется в виду нуль-символ, которым строка заканчивается), необходимости явно указывать размер строки нет.

Учитывая все вышеозначенное, легко предугадать результат выполнения программы: на экране появится сообщение `Length is 16`. Причем вычисляется именно длина строки-значения массива: в строке `"This is a string"` (с учетом пробелов) 16 символов, а массив `str` состоит из 20 элементов.

Аргументы функции `main()`

У главного метода программы `main()` могут быть аргументы: число параметров командной строки и массив с текстовыми значениями этих параметров. Обычно параметры называют `argc` (размер массива) и `argv` (символьный двумерный массив), однако это не обязательно. В листинге 4.15 приведен пример программы, в которой в столбик выводятся на экран все переданные в командной строке параметры функции. При этом первым и всегда присутствующим параметром является название запускаемой программы (полный путь к файлу).

Листинг 4.15. Аргументы главного метода программы

```
include <iostream>
using namespace std;
int main(int size,char *str[]){
    int i;
    for(i=0;i<size;i++){
        cout<<i+1<<"-th argument is: "<<str[i]<<endl;
    }
    return 0;
}
```

Первый целочисленный `size` аргумент особых вопросов не вызывает. Второй аргумент, объявленный как `char *str[]` – массив, элементами которого являются текстовые строки, реализованные в виде символьных массивов. Более подробно о работе с такими массивами речь будет идти в главе 5. В данном случае важно лишь то, что ссылка `str[i]` означает *i*-ю строку, т.е. текстовое значение *i*-го параметра командной строки. В приведенном программном коде индексная переменная *i* пробегает значения в соответствии с размером массива (от 0 до `size-1` включительно). Для каждого значения этого индекса на экран выводится соответствующий параметр командной строки, для чего используется ссылка `str[i]`. Например, если после компиляции файл программы называется `mytest.exe`, размещен в директории `C:\Program Files\MyProgs` и запускается на выполнение командой `mytest.exe Alexei Vasilev 2008`, в результате получим:

```
1-th argument is: C:\Program Files\MyProgs\mytest.exe
2-th argument is: Alexei
3-th argument is: Vasilev
4-th argument is: 2008
```

Если параметры в командную строку передавать не планируется, аргументы для метода `main()` можно не указывать (а можно указывать).

Аргументы по умолчанию

Курс у нас один – правильный.

В.С. Черномырдин

Для аргументов функций можно указывать значения по умолчанию. **Если аргумент имеет значение по умолчанию, то в случае, если при вызове функции этот аргумент явно не указан, используется его значение по умолчанию.**

Чтобы задать аргументу значение по умолчанию, в списке аргументов функции после имени этого аргумента через знак равенства указывается соот-

ветствующее значение, т.е. синтаксис определения значения по умолчанию в прототипе функции следующий:

```
тип имя_функции(тип аргумент=значение) {код функции}
```

Если у функции несколько аргументов, то значения по умолчанию можно задавать для любого их количества, в том числе и для всех. При этом аргументы, которые имеют значения по умолчанию, должны следовать в списке аргументов функции последними.

В случае, когда прототип функции указывается до ее определения, значения по умолчанию указываются только в прототипе функции. В листинге 4.16 приведен пример использования функций с аргументами, имеющими значения по умолчанию.

Листинг 4.16. Аргументы со значениями по умолчанию

```
#include <iostream>
using namespace std;
//Аргумент функции имеет значение по умолчанию:
void showX(int x=0){
    cout<<"x = "<<x<<endl;
}
//Два аргумента функции в прототипе имеют значение по
//умолчанию,
//сама функция описана в конце программы:
void showXYZ(int x,int y=1,int z=2);
int main(){
    showX(3);
    showX();
    showXYZ(4,5,6);
    showXYZ(7,8);
    showXYZ(9);
    return 0;
}
//При описании функции значения по умолчанию не указываются:
void showXYZ(int x,int y,int z){
    cout<<"x = "<<x<<" ";
    cout<<"y = "<<y<<" ";
    cout<<"z = "<<z<<endl;
}
```

В программе объявлены две функции: у функции `showX()` один аргумент с целочисленным аргументом, имеющим значение по умолчанию, а у функции `showXYZ()` из трех целочисленных аргументов значения по умолчанию указаны для двух. Действие функций состоит в том, что выводятся значения аргументов. В главном методе приведены вызовы этих функций

с передачей разного числа аргументов: первая функция `showX()` вызывается с одним аргументом и без аргумента, а вторая функция `showXYZ()` вызывается с числом аргументов от одного до трех. Результат выполнения программы будет иметь вид

```
x=3
x=0
x=4 y=5 z=6
x=7 y=8 z=2
x=9 y=1 z=2
```

В прототипе функции, как отмечалось, формальные значения для аргументов могут не указываться (но они указываются в описании функции). Если аргументы имеют значения по умолчанию, знак равенства и значение для соответствующего аргумента в прототипе указывают сразу после идентификатора типа аргумента. Например, в приведенном листинге 6.16 вместо прототипа `void showXYZ(int x, int y=1, int z=2)` можно было использовать прототип `void showXYZ(int x, int=1, int=2)`. Функциональность программы от этого не изменится.

Возвращение функцией указателя

Функция в качестве значения может возвращать указатель. Главное, что для этого нужно сделать, – предусмотреть соответствующие инструкции в программном коде функции. В прототипе функции, возвращающей в качестве значения указатель, перед именем функции указывают оператор `*`. Пример функции, возвращающей указатель, приведен в листинге 4.17.

Листинг 4.17. Функция возвращает в качестве значения указатель

```
#include <iostream>
using namespace std;
int *mpoint(int &n, int &m){
    if(n>m) return &n;
    else return &m;
}
int main(){
    int n=3, m=5;
    int *p;
    p=mpoint(n, m);
    (*p)++;
    cout<<"n ="<<n<<endl;
    cout<<"m ="<<m<<endl;
    return 0;
}
```

В программе объявляется функция `mpoint()`, возвращающая в качестве результата указатель на максимальный из двух ее аргументов. Сразу отметим, что оба аргумента передаются по ссылке – если бы они передавались по значению, не было бы смысла возвращать указатель на один из аргументов, поскольку в этом случае указатель ссылался бы на временную переменную-копию аргумента.

При возвращении в качестве значения указателя не следует забывать, что возвращается адрес большего из аргументов, а не сам аргумент. Адрес переменной (аргумента) получаем, указав перед именем соответствующей переменной оператор `&`, что и было сделано в командах `return &n` и `return &m`.

В главном методе программы объявляются две целочисленные переменные `n` и `m` со значениями 3 и 5 соответственно. Именно они передаются аргументами функции `mpoint()`. Результат записывается в переменную-указатель `p`. В результате переменная `p` в качестве значения получает адрес переменной `m`. Поэтому после выполнения команды `(*p)++` значение переменной `m` увеличивается на единицу. Результатом выполнения программы станет пара строк

```
n =3
m =6
```

В конечном счете указатель – это всего лишь переменная, значением которой является адрес памяти. Нет ничего удивительного в том, что адрес возвращается функцией.

Возвращение функцией ссылки

В C++ функции могут возвращать в качестве результата ссылку на значение. Для того чтобы функция возвращала ссылку на значение, перед именем функции в ее прототипе (и при описании) необходимо использовать оператор `&`. Пример функции, результатом которой является ссылка, приведен в листинге 4.18.

Листинг 4.18. Функция возвращает в качестве значения ссылку

```
#include <iostream>
using namespace std;
int &mpoint(int &n,int &m){
    if(n>m) return n;
    else return m;
}
int main(){
    int n=3,m=5;
```



```

int p;
mpoint(n,m)=2;
p=mpoint(n,m);
cout<<"n ="<<n<<endl;
cout<<"m ="<<m<<endl;
cout<<"p ="<<p<<endl;
return 0;
}

```

Приведенный программный код представляет собой модификацию предыдущего примера из листинга 4.17, только в данном случае функцией `mpoint()` возвращается не указатель, а ссылка на больший из своих аргументов. При объявлении функции перед названием указан оператор `&` (признак того, что функцией возвращается ссылка). В качестве значения, как отмечалось, возвращается больший из аргументов `n` и `m` (инструкции `return n` и `return m` в условном операторе). Однако в силу того, что функция объявлена как ссылка, в действительности возвращается не значение соответствующей переменной, а ссылка на нее! Драматичность этого утверждения легко подтвердить с помощью кода, представленного в главном методе программы.

В методе `main()`, как и в предыдущем примере, инициализируются две целочисленные переменные `n=3` и `m=5`, а также объявляется целочисленная переменная `p`. Далее следует на первый взгляд довольно странная команда `mpoint(n,m)=2`, после чего командой `p=mpoint(n,m)` присваивается значение переменной `p`. Но самое интересное – это результат, который получаем при выполнении этой программы:

```

n =3
m =2
p =3

```

По крайней мере, странно выглядят значения переменных `m` и `p`. Разумеется, это не совсем так, а точнее, совсем не так. Начнем с команды `mpoint(n,m)=2`. Чтобы понять смысл этой команды, вспомним, что результатом инструкции `mpoint(n,m)` является ссылка на больший из аргументов – при текущих значениях аргументов `n` и `m` это есть ссылка на переменную `m`. Поэтому команда `mpoint(n,m)=2` эквивалентна присваиванию переменной `m` значения 2. Далее, командой `p=mpoint(n,m)` переменной `p` присваивается значение максимального из аргументов `n` и `m` – теперь это переменная `n` (поскольку на момент вызова означенной команды значение `n` равно 3, а значение `m` равно 2). Таким образом, переменная `n` остается со значением 3, такое же значение имеет переменная `p`, а значение переменной `m` равно 2. Отметим также, что аргументы функции `mpoint()` передаются по ссылке по тем же причинам, что и в предыдущем примере.

Указатели на функции

Чем фундаментальнее закономерность, тем проще ее можно сформулировать.

II. Капитула

Это может показаться на первый взгляд странным, но указатель может ссылаться на функцию. Дело в том, что каждая функция хранится в памяти, соответствующая область памяти имеет адрес, и этот адрес можно записать в переменную-указатель на функцию. Вызов функции осуществляется через адрес, по которому она записана. Этот адрес также называют точкой входа в функцию.

Главное правило, которое следует запомнить, состоит в том, что **имя функции (без круглых скобок и аргументов) является указателем на функцию**. Значение этого указателя есть адрес, по которому записана функция.

Указатель на функцию объявляется следующим образом. Сначала указывается тип результата, который возвращается соответствующей функцией, затем заключенные в круглые скобки оператор * и имя указателя, а после этих круглых скобок еще одни круглые скобки с перечислением типов аргументов функции. Например, если функцией в качестве значения возвращается значение типа `int` и у нее два аргумента типа `double` и `char`, то указатель `p` на эту функцию объявляется как `int (*p)(double, char)`.

В качестве значения такому указателю присваивается имя функции, на которую должен ссылаться указатель. Пример использования указателя на функцию приведен в листинге 4.19.

Листинг 4.19. Указатели на функции

```
#include <iostream>
using namespace std;
//Функция возведения в квадрат:
double sqr(double x){
    return x*x;}
//Функция возведения в куб:
double cube(double x){
    return x*x*x;}
//Функция со вторым аргументом-указателем на функцию:
void myfunc(double x,double (*f)(double)){
    cout<<f(x)<<endl;}
int main(){
    double z;
    //Указатель на функцию:
    double (*p)(double);
    cout<<"z = ";
```

```

cin>>z;
//Указателю присваивается значение:
p=cube;
//Использование указателя и имени функции:
myfunc(z,sqr);
myfunc(z,p);
cout<<p(z)<<endl;
//Адрес функции:
cout<<sqr<<endl;
cout<<cube<<endl;
cout<<p<<endl;
return 0;
}

```

В программе объявляются три функции. У функции `sqr()` аргумент типа `double`, результатом является квадрат аргумента – число типа `double`. Функцией `cube()` в качестве значения возвращается значение типа `double` – куб аргумента, который также имеет тип `double`. У функции `myfunc()` два аргумента. Первый аргумент имеет тип `double`, а второй аргумент является указателем на функцию. Он объявлен как `double (*f)(double)`. В данном случае `f` – формальное название аргумента. Оператор `*` означает, что это указатель. Ключевое слово `double` в круглых скобках – тип аргумента функции, а `double` перед именем указателя – тип результата функции. Таким образом, функции `myfunc()` первым аргументом передается число, а вторым – имя функции.

В результате вызова функции отображается значение `f(z)`, то есть значение функции, имя которой указано вторым аргументом, от аргумента – первого параметра функции `myfunc()`.

В главном методе программы командой `double (*p)(double)` объявляется указатель на функцию. Значение указателю присваивается командой `p=cube`. После этого указатель `p` ссылается на функцию `cube()`. Далее этот указатель и имена функций используются в командах `myfunc(z,sqr)` (квадрат числа `z`), `myfunc(z,p)` (куб числа `z`) и `p(z)` (куб числа `z`). В конце программы разными способами отображаются адреса функций с использованием указателя `p` и имен функций. Результат выполнения программы может иметь вид (жирным шрифтом выделены вводимые пользователем данные):

```

z = 5
25
125
125
00401195
0040128F
0040128F

```

Еще раз обращаем внимание читателя, что имя функции является адресом области памяти, в которую записана функция. В этом отношении ситуация напоминает ситуацию с массивами – имя массива является указателем на первый элемент.

Рекурсия

Мой соперник не будет избран, если дела не пойдут хуже. А дела не пойдут хуже, если его не выберут.

Дж. Буш (старший)

Под рекурсией подразумевают вызов в теле функции этой же самой функции. Рекурсивный вызов может быть как прямым (функция вызывается в теле этой же функции), так и косвенным (в функции вызываются другие функции, в теле которых, в свою очередь, вызывается исходная функция). Обычно к такому приему прибегают, когда программируемая последовательность действий может быть сформулирована в терминах рекурсивной зависимости как последовательность значений, каждое из которых определяется на основе предыдущего по одной и той же схеме или принципу. В качестве примера использования рекурсии рассмотрим программу по вычислению факториала числа. Напомним, что факториал числа определяется как произведение всех натуральных чисел от единицы до этого числа включительно: $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 1) \cdot n$. В силу этого определения можем представить факториал числа n в виде произведения факториала числа $(n - 1)$ на число n , а именно: $n! = n \cdot (n - 1)!$. Этим соотношением воспользуемся для создания специальной функции в C++ (листинг 4.20).

Листинг 4.20. Функция для вычисления факториала числа с использованием рекурсии

```
#include <iostream>
using namespace std;
int factorial(int n){
    if(n==1) return 1;
    else return n*factorial(n-1);
}
int main(){
    int n;
    cout<<"Enter n = ";
    cin>>n;
    cout<<"n! = "<<factorial(n)<<endl;
    return 0;
}
```

Целочисленная функция `factorial()` достаточно простая: у нее всего один целочисленный аргумент `n`, а тело кода состоит из условного оператора. Проверяемым является условие `n==1`, при выполнении которого возвращается значение 1 (следствие того замечательного факта, что факториал единицы равен единице, то есть $1! = 1$). Если аргумент у функции не единичный, значение вычисляется в виде выражения `n*factorial(n-1)`. В этом выражении вызывается определяемая функция, но с уменьшенным на единицу аргументом.

При вычислении значения функции возможны два варианта: аргумент единичный и аргумент больше единицы (другие варианты не рассматриваем в принципе). Если аргумент единичный – все просто. В качестве значения возвращается число 1. В противном случае для вычисления значения функции вызывается эта же функция, но только аргумент у нее на единицу меньше. Если после уменьшения на единицу аргумент все равно не единичный, снова вызывается функция, и ее аргумент еще на единицу меньше и т.д. до тех пор, пока аргумент у вызываемой функции не станет единичным. Для единичного аргумента значение вычисляется «в лоб», после чего вся эта цепочка последовательных вызовов раскручивается в обратном порядке.

Отметим, что, несмотря на эффективность программного кода с рекурсивным вызовом, эффективностью он отличается редко, поскольку требует использования существенных системных ресурсов.

Перегрузка функций

Нам никто не мешает перевыполнить наши законы.

В.С. Черномырдин

Функции в C++ обладают замечательным свойством – их можно перегружать. **Под перегрузкой подразумевают создание и использование функций с разными прототипами, но одинаковыми названиями.** Поскольку название функции является частью ее прототипа, становится понятно, что отличаются прототипы не названием, а числом и типом аргументов и типом возвращаемого результата. Причем вполне достаточно хотя бы одного отличия.

Перегрузка функций – инструмент мощный и весьма полезный. Поэтому любой уважающий себя программист должен владеть им в совершенстве. Обычно перегрузку используют в тех случаях, когда приходится выполнять однотипные действия с разными объектами или разными типами данных.

При перегрузке функции, фактически, создается несколько функций с одинаковыми названиями, которые, однако, можно различить по остальным их атрибутам. При вызове презагруженной функции в программе выбор нужного варианта функции осуществляется исходя из использованного синтаксиса вызова функции. В листинге 4.21 приведен пример перегрузки функции.

Листинг 4.21. Перегрузка функции

```
#include <iostream>
using namespace std;
//Первый вариант функции:
void showArgs(double x){
    cout<<"Double-number "<<x<<endl;
}
//Второй вариант функции:
void showArgs(double x,double y){
    cout<<"Double-numbers "<<x<<" and "<<y<<endl;
}
//Третий вариант функции:
void showArgs(char s){
    cout<<"Symbol "<<s<<endl;
}
//Четвертый вариант функции:
int showArgs(int n){
    return n;
}
int main(){
    int n=3;
    double x=2.5,y=5.1;
    char s='w';
    //Первый вариант функции:
    showArgs(x);
    //Второй вариант функции:
    showArgs(x,y);
    //Третий вариант функции:
    showArgs(s);
    //Четвертый вариант функции:
    cout<<"Int-number "<<showArgs(n)<<endl;
    return 0;
}
```

В программе описано четыре варианта функции `showArgs()`, назначение которой состоит в выводе на экран информации о ее аргументах. Однако в зависимости от количества и типа аргументов выполняются немного разные действия. В частности, предусмотрены такие варианты передачи аргументов:

1. Один аргумент типа `double`
2. Два аргумента типа `double`
3. Один аргумент типа `char`
4. Один аргумент типа `int`

В первых трех случаях результат функцией не возвращается (тип функции `void`), а на экран выводится сообщение о типе и значении аргумента (или аргументов). В четвертом случае функцией в качестве значения возвращается аргумент.

В главном методе программы функция `showArgs()` вызывается в разном контексте: вызов `showArgs(x)` соответствует варианту функции для одного аргумента типа `double`, вызов `showArgs(x, y)` соответствует варианту функции для двух аргументов типа `double`, вызов `showArgs(s)` соответствует варианту функции для одного аргумента типа `char`, вызов `cout<<"Int-number " <<showArgs(n) <<endl` соответствует варианту функции с одним аргументом типа `int`. Результат выполнения программы будет следующим:

```
Double-number 2.5
Double-numbers 2.5 and 5.1
Symbol w
Int-number 3
```

При работе с переопределенными функциями не следует забывать об автоматическом приведении типов. Эти два механизма в симбиозе могут давать очень интересные, а иногда и странные результаты. Рассмотрим пример. Так, в листинге 4.22 приведен исходный код программы, не содержащий перегруженных функций.

Листинг 4.22. Автоматическое приведение типов

```
#include <iostream>
using namespace std;
void ndiv(double x, double y){
    cout<<"x/y= " <<x/y<<endl;
}
int main(){
    double x=10,y=3;
    int n=10,m=3;
    ndiv(x,y);
    ndiv(n,m);
    return 0;
}
```

Результат выполнения такой программы будет следующим:

```
x/y= 3.33333
x/y= 3.33333
```

Первая строка, которая является результатом выполнения команды `ndiv(x, y)`, думается, вопросов не вызывает. Результатом вызова функции `ndiv()` является частное двух чисел – аргументов функции. Хотя в команде `ndiv(n, m)` аргументами функции указаны целые числа, благодаря автоматическому приведению типов целочисленные значения расширяются до типа `double`, после чего вычисляется нужный результат.

Однако стоит изменить программный код, перегрузив функцию `ndiv()`, как показано в листинге 4.23, и результат изменится кардинально.

Листинг 4.23. Перегрузка и автоматическое приведение типов

```
#include <iostream>
using namespace std;
void ndiv(double x, double y){
    cout<<"x/y= "<<x/y<<endl;
}
void ndiv(int n, int m){
    cout<<"n/m= "<<n/m<<endl;
}
int main(){
    double x=10,y=3;
    int n=10,m=3;
    ndiv(x,y);
    ndiv(n,m);
    return 0;
}
```

В частности, на экране в результате выполнения программы появится сообщение:

```
x/y= 3.33333
n/m= 3
```

Дело в том, что измененный программный код содержит перегруженную функцию `ndiv()`, для которой предусмотрен вызов как с двумя аргументами типа `double`, так и с двумя аргументами типа `int`. В последнем случае, хотя результат функции формально вычисляется так же, как и в исходном варианте, для целых чисел оператор `/` означает целочисленное деление с отбрасыванием остатка. Поэтому в результате получаем число 3, а не 3.33333, как для вызова функции с `double`-аргументами. Более того, нередко случаются ситуации, когда с учетом автоматического приведения ти-

пов невозможно однозначно определить, какой из вариантов перегруженной функции необходимо вызывать. Например, если для перегруженной функции предусмотрены два варианта передачи аргумента для типа данных `float` и для типа данных `double`, то передача функции в качестве аргумента `int`-переменной приведет к ошибке компиляции, поскольку непонятно, к какому формату данных (`float` или `double`) нужно преобразовывать тип `int`. Такие ситуации относятся к разряду логических ошибок, и их нужно внимательно отслеживать.

Как и обычные функции, перегруженные функции могут иметь аргументы со значениями, используемыми по умолчанию. Причем для каждого варианта перегружаемой функции эти значения по умолчанию могут быть разными. Единственное, за чем необходимо постоянно следить, – чтобы наличие значений по умолчанию у аргументов не приводило к неоднозначным ситуациям при вызове перегруженной функции. Ситуацию иллюстрирует пример в листинге 4.24.

Листинг 4.24. Перегрузка и значения по умолчанию

```
#include <iostream>
using namespace std;
void hello(){
    cout<<"Hello, my friend!\n";
}
void hello(char str[],char name[]="Alex"){
    cout<<str<<"", "<<name<<"!"<<endl;
}
int main(){
    hello();
    hello("Hello","Peter");
    hello("Hi");
    return 0;
}
```

У перегруженной функции `hello()` два варианта: без аргументов и с двумя аргументами-массивами типа `char`. Причем в последнем случае второй аргумент имеет значение по умолчанию.

Если функция вызывается без аргументов, в результате ее выполнения на экран выводится сообщение `Hello, my friend!`. При вызове функции с двумя аргументами (каждый аргумент – текстовая строка, реализованная в виде массива символов) на экран последовательно выводятся текстовые значения аргументов функции. Поскольку второй аргумент имеет значение по умолчанию `Alex`, то формально функцию можно вызывать и с одним текстовым аргументом. В главном методе функция `hello()` последова-

тельно вызывается без аргументов (команда `hello()`), с двумя аргументами (команда `hello("Hello", "Peter")`) и с одним аргументом (команда `hello("Hi")`). Результат выполнения программы выглядит следующим образом:

```
Hello, my friend!
Hello, Peter!
Hi, Alex!
```

Но если попытаться для второго варианта функции (с двумя аргументами) указать значение по умолчанию и для первого аргумента, возникнет ошибка. Причина в том, что наличие у каждого из аргументов функции `hello()` значения по умолчанию приводит к неоднозначной ситуации: если функция при вызове указана без аргументов, невозможно определить, вызывается ли это вариант функции без аргументов, или нужно использовать вариант функции с двумя аргументами со значениями по умолчанию. Как уже отмечалось, на подобные ситуации следует постоянно обращать внимание при создании программных кодов в процессе загрузки функций.

Примеры решения задач

В этом разделе рассматриваются примеры решения задач, в которых основу алгоритма составляют описываемые пользователем функции. Некоторые задачи повторяют те, что рассматривались в предыдущих главах, однако теперь соответствующие задачи решаются путем создания функции (или функций) пользователя.

■ Функция для вычисления гиперболического синуса

Напишем программу, в которой создается функция для вычисления гиперболического синуса по формуле $sh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}$.

В программе определено два варианта функции – т.е. функция перегружена. В первом варианте аргументом функции является переменная x , а граница ряда определяется как константа. Во втором случае количество слагаемых ряда передается вторым аргументом функции. Соответствующий программный код приведен в листинге 4.25.

Листинг 4.25. Синус гиперболический

```

#include <iostream>
#include <cmath>
using namespace std;
//Верхний предел суммы по умолчанию:
const int N=100;
//Функция с одним аргументом:
double sh(double z){
    int n;
    double s=z,q=z;
    for(n=1;n<=N;n++){
        q*=z*z/(2*n)/(2*n+1);
        s+=q;}
    return s;}
//Функция с двумя аргументами:
double sh(double z,int m){
    int n;
    double s=z,q=z;
    for(n=1;n<=m;n++){
        q*=z*z/(2*n)/(2*n+1);
        s+=q;}
    return s;}
int main(){
    //Аргумент для функции:
    double x;
    //Индексная переменная и предел для суммы:
    int i,m=9;
    //Ввод аргумента:
    cout<<"Enter x = ";
    cin>>x;
    //Значения ряда для разного числа слагаемых:
    for(i=1;i<=m;i++){
        cout<<i<<" : sh("<<x<<" = "<<sh(x,i)<<endl;}
    cout<<"-----\n";
    //Верхняя граница индексной переменной ряда равна N:
    cout<<N<<" : sh("<<x<<" = "<<sh(x)<<endl;
    //Вызов встроенной функции:
    cout<<"Test value: "<<sinh(x)<<endl;
    return 0;}

```

В главном методе программы с помощью оператора цикла вызывается функция с явным указанием верхней границы ряда (число слагаемых в ряде на единицу больше). Далее вызывается вариант функции с одним

аргументом и, кроме этого, значение гиперболического синуса вычисляется с помощью встроенной функции. Результат может выглядеть так, как показано ниже:

```
Enter x = 5
1 : sh(5) = 25.8333
2 : sh(5) = 51.875
3 : sh(5) = 67.376
4 : sh(5) = 72.7583
5 : sh(5) = 73.9815
6 : sh(5) = 74.1776
7 : sh(5) = 74.2009
8 : sh(5) = 74.203
9 : sh(5) = 74.2032
-----
100: sh(5) = 74.2032
Test value = 74.2032
```

Чем меньше аргумент гиперболического синуса, тем меньше слагаемых нужно брать при вычислении соответствующего ряда. В приведенном примере при аргументе $x = 5$ уже при значении верхней границы индекса суммирования $m = 9$ вычисляется такое же значение, как и с помощью встроенной функции.

Отметим, что в данном случае вместо перегрузки функции можно было воспользоваться механизмом определения значения второго аргумента по умолчанию.

■ Вычисление произведения

Практически так же можно определить функцию для вычисления произведений. В данном случае воспользуемся рекурсией. Пример программного кода с использованием рекурсивной функции для вычисления произведения

$$\prod_{n=0}^{\infty} (1 + x^{2^n}) = \frac{1}{1-x} \quad (\text{аргумент } |x| < 1)$$

приведен в листинге 4.26.

Листинг 4.26. Вычисление произведения

```
#include <iostream>
#include <cmath>
using namespace std;
//Функция (с рекурсией) для вычисления произведения:
double MyProd(double x, int N) {
    if (N > 0) return MyProd(x, N-1) * (1 + pow(x, pow(2, N)));
    else return 1 + x; }
```

```
int main(){
    //Аргументы функции:
    double x;
    int N;
    //Ввод аргумента:
    cout<<"Enter x = ";
    cin>>x;
    cout<<"Enter N = ";
    cin>>N;
    cout<<"Product value: "<<MyProd(x,N)<<endl;
    //Проверка результата:
    cout<<"Test value: "<<1/(1-x)<<endl;
    return 0;}
```

Следует учесть, что показатель степени в множителях произведения растет с ростом индексной переменной n как 2^n , поэтому большими значениями верхней границы произведения увлекаться не стоит. Типичный пример выполнения программы имеет вид

```
Enter x = 0.2
Enter N = 10
Product value: 1.25
Test value: 1.25
```

Видим, что даже небольшое количество слагаемых дает формально точный результат (с учетом точности округления).

■ Метод половинного деления

Уравнения вида $f(x) = 0$ могут решаться методом половинного деления. Корень ищется на интервале $x \in (a, b)$, таком что на границах интервала функция принимала значения разных знаков (что означает необходимость выполнения условия $f(a)f(b) < 0$). После того, как указан интервал поиска решения, проверяется значение функции $f(c)$ в центральной точке $c = \frac{a+b}{2}$. В эту точку переносится та граница интервала, знак функции

на которой совпадает со знаком функции в центральной точке. Таким образом, интервал поиска уменьшается в два раза. Продолжая процесс необходимое количество раз, добиваемся нужной точности вычисления корня уравнения.

В листинге 4.27 приведен пример кода, в котором методом половинного деления реализовано решение уравнения $x^2 - 9x + 14 = 0$ (корни $x = 2$ и $x = 7$). При этом не только зависимость $f(x) = x^2 - 9x + 14$ реализована в виде функции, но и сама процедура поиска корня.

Листинг 4.27. Метод половинного деления

```

#include <iostream>
using namespace std;
//Функция для полинома:
double F(double x){
    return x*x-9*x+14;}
//Функция поиска корня:
double FindRoot(double (*f)(double),double a,double b,double eps){
    double c;
    while((b-a)/2>eps){
        c=(a+b)/2;
        if((f(a)*f(c))>0) a=c;
        else b=c;
    }
    return c;}
int main(){
    //Интервал, погрешность и корень:
    double a,b,eps,x;
    cout<<"interval: ";
    cin>>a;
    cin>>b;
    //Проверка корректности интервала:
    if(F(a)*F(b)>0){
        cout<<"Wrong interval!\n";
        return 0;} cout<<"error: ";
    cin>>eps;
    //Поиск решения:
    x=FindRoot(F,a,b,eps);
    cout<<"x = "<<x<<endl;
    return 0;}

```

При описании функции FindRoot () при передаче аргументов используетсЯ ссылка на функцию. Это позволяет использовать одну и ту же функцию для решения различных уравнений – достаточно только указать первым аргументом имя функции, определяющей решаемое уравнение. Второй и третий аргументы функции – левая и правая границы интервала поиска решения. Последний, четвертый аргумент – погрешность, с которой необходимо вычислить корень.

Пример выполнения программы при вычислении корня $x = 7$ может иметь вид:

```

interval: 3 10
error: 0.0001
x = 6.99998

```

Если интервал поиска решения введен некорректно, получим следующий результат:

```
interval: 3 6
Wrong interval!
```

Близок по своей идеологии к методу половинного деления метод хорд.

■ Метод хорд

От рассмотренного выше случая в методе хорд по-иному выбирается точка, в которой проверяется значение функции. В частности, через граничные точки графика функции $f(x)$ (решается уравнение $f(x) = 0$) проводится хорда. В качестве контрольной точки выбирается точка пересечения этой хорды с координатной осью. Во всем остальном алгоритм такой же, как и в методе половинного деления. При поиске корня на интервале $x \in (a, b)$ контрольная точка внутри этого интервала выбирается как $c = \frac{f(b)a - f(a)b}{f(b) - f(a)}$.

В листинге 4.28 реализован программный код, с помощью которого решает-ся все то же уравнение $x^2 - 9x + 14 = 0$, но уже методом хорд.

Листинг 4.28. Метод хорд

```
#include <iostream>
#include<cstdlib>
using namespace std;
//Функция для полинома:
double F(double x){
    return x*x-9*x+14;}
//Функция поиска корня:
double FindRoot(double (*f)(double),double a,double b,double eps){
    double c;
    while(abs(f(b)-f(a))>eps){
        c=(f(b)*a-f(a)*b)/(f(b)-f(a));
        if((f(a)*f(c))>0) a=c;
        else b=c;
    }
    return c;}
int main(){
    //Интервал, погрешность и корень:
    double a,b,eps,x;
    cout<<"interval: ";
    cin>>a;
    cin>>b;
    //Проверка корректности интервала:
    if(F(a)*F(b)>0){
```

```

    cout<<"Wrong interval!\n";
    return 0;} cout<<"error: ";
cin>>eps;
//Поиск решения:
x=FindRoot(F,a,b,eps);
cout<<"x = "<<x<<endl;
return 0;}

```

В результате выполнения программы получаем

```

interval: 3 10
error: 0.0001
x = 7

```

Обращаем внимание, что в данном случае погрешность eps определяет не точность поиска корня, а точность выполнения соотношения $f(x) = 0$ (точнее, ширину интервала разности значений функции на интервале поиска). С практической точки зрения метод хорд обеспечивает более быструю сходимость по сравнению с методом половинного деления.

■ Метод Ньютона и перегрузка функции вычисления корня

Метод Ньютона уже рассматривался ранее при решении уравнений полиномиального типа. Здесь остановимся на реализации этого метода в том случае, если решаемое уравнение представлено в программе в виде функции. Особенность подхода состоит в том, что производная, значение которой необходимо знать для вычисления нового приближения для корня, рассчитывается в численном виде.

Напомним, что в методе Ньютона указывается начальное приближение для корня уравнения $f(x) = 0$, после чего итерационным образом этот корень начинает уточняться в соответствии с соотношением $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

Пример программы, в которой использован метод Ньютона для решения уравнения $x^2 - 9x + 14 = 0$, приведен в листинге 4.29.

Листинг 4.29. Метод Ньютона

```

#include <iostream>
using namespace std;
const int N=20;
//Функция для полинома:
double F(double x){
    return x*x-9*x+14;}

```



```
//Функция поиска корня:
double FindRoot(double (*f)(double),double x0,int n){
    double x=x0,df,h=0.00001;
    df=(f(x+h)-f(x))/h;
    for(int i=1;i<=n;i++){
        x=x-f(x)/df;
    }
    return x;}

int main(){
    //Начальное приближение и корень:
    double x0,x;
    cout<<"initial x0 = ";
    cin>>x0;
    //Поиск решения:
    x=FindRoot(F,x0,N);
    cout<<"x = "<<x<<endl;
    return 0;}
```

Ниже представлен пример выполнения программы:

```
initial x0 = 12
x = 7.00048
```

Обращаем внимание читателя на способ вычисления производной для функции уравнения командой $df = (f(x+h) - f(x)) / h$. Здесь фактически использована разностная схема для определения производной функции

$f(x)$ в виде $f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$, где шаг изменения аргумента h

является малой величиной.

Отметим также, что на практике при программировании методов решения уравнений и прочих подобных задач используется перегрузка функций, что позволяет использовать, в зависимости от ситуации, различные алгоритмы вычисления корня. Пример такой перегрузки можно найти в листинге 4.30.

Листинг 4.30. Перегрузка функции вычисления корня

```
#include <iostream>
using namespace std;
const int N=20;
//Функция для полинома:
double F(double x){
    return x*x-9*x+14;}
//Функция поиска корня (метод Ньютона):
double FindRoot(double (*f)(double),double x0){
    double x=x0,df,h=0.00001;
```

```

    df=(f(x+h)-f(x))/h;
    for(int i=1;i<=N;i++)
        x=x-f(x)/df;
    return x;}
//Функция поиска корня (метод хорд):
double FindRoot(double (*f)(double),double a,double b){
    double c;
    for(int i=1;i<=N;i++){
        c=(f(b)*a-f(a)*b)/(f(b)-f(a));
        if((f(a)*f(c))>0) a=c;
        else b=c;
    }
    return c;}
int main(){
    //Начальное приближение, интервал и корни уравнения:
    double x0,a,b,x1,x2;
    cout<<"initial x0 = ";
    cin>>x0;
    cout<<"interval: ";
    cin>>a;
    cin>>b;
    //Поиск решения:
    x1=FindRoot(F,x0);
    x2=FindRoot(F,a,b);
    cout<<"Newton method: x = "<<x1<<endl;
    cout<<"Chord method: x = "<<x2<<endl;
    return 0;}

```

По сравнению с предыдущими случаями, алгоритмы реализации метода хорд и метода Ньютона немного изменены: соответствующие итерационные процедуры выполняются определенное количество раз (определяется целочисленной константой N), а перегруженная функция FindRoot() утратила свой последний аргумент – теперь она может принимать два или три аргумента. Пример выполнения программы приведен ниже:

```

initial x0 = 12
interval: 3 12
Newton method: x = 7.00048
Chord method: x = 6.99998

```

Если у функции FindRoot() при вызове два аргумента, используется метод Ньютона. Если функции передано три аргумента – используется метод хорд.

■ Вывод строки

Рекурсия может использоваться в достаточно неожиданных ситуациях. В листинге 4.31 приведена программа с функцией для вывода в инверсном порядке строки, переданной аргументом этой функции. В функции с помощью рекурсии реализован следующий алгоритм. Если текущий элемент соответствующего символьного массива не является нуль-символом и не совпадает с символом `i`, вызывается функция для обработки следующего символа массива. Если элемент массива является символом `i`, работа функции заканчивается. Если элемент массива является нуль-символом окончания строки, завершается работа программы с сообщением `I didn't find any 'i'!`. Таким образом, если строка-аргумент функции содержит хотя бы одну литеру `i`, строка распечатывается в обратном порядке от этого символа (первого, если их несколько) до начала строки. Если символов `i` в строке нет, выводится сообщение `I didn't find any 'i'!`.

Листинг 4.31. Вывод строки в инверсном порядке

```
#include <iostream>
using namespace std;
void ShowStr(char *str){
    if(*str!='i'&&*str) ShowStr(str+1);
    else if(*str=='i') return;
    else {cout<<"I didn't find any 'i'!\n";
        exit(0);}
    cout<<*str;
}
int main(){
    char s[80];
    cout<<"Enter text here: ";
    gets(s);
    ShowStr(s);
    cout<<endl;
    return 0;}
```

Результат выполнения программы может иметь следующий вид:

```
Enter text here: Hello, my dear friend!
rf raed ym ,olleH
```

Здесь ввод пользователя выделен жирным шрифтом. Если символа `i` во введенной пользователем строке нет, результат будет следующим:

```
Enter text here: Hello, World!
I didn't find any 'i'!
```

Отметим, что несмотря на некоторую эффектность приведенного выше кода, того же результата можно было добиться и более простыми средствами, без применения рекурсии.

■ Вычисление статистических характеристик

Рассмотрим пример создания специальных функций для вычисления статистических характеристик на основе данных массива, переданного аргументом функции. В листинге 4.32 приведен пример программы, в которой создано несколько перегруженных функций: для вычисления среднего значения, для вычисления дисперсии и для отображения указанных статистических характеристик. В последнем случае в соответствующей функции вызываются функции вычисления означенных показателей. Функции вычисления статистических характеристик перегружены так, что позволяют, кроме стандартных вычислений, проводить соответствующие вычисления по подмножествам.

Листинг 4.32. Статистические характеристики

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
//Среднее по всему массиву:
double avr(double *x,int n){
    int i;
    double s=0;
    for(i=0;i<n;i++) s+=x[i];
    s/=n;
    return s;
}
//Среднее по части массива:
double avr(double *x,int n1,int n2){
    int i,n=n2-n1+1;
    double s=0;
    for(i=n1-1;i<n2;i++) s+=x[i];
    s/=n;
    return s;
}
//Дисперсия по массиву:
double dev(double *x,int n){
    double x0,s=0;
    int i;
    x0=avr(x,n);
    for(i=0;i<n;i++) s+=x[i]*x[i];
    s/=n;
    return s-x0*x0;
}
//Дисперсия по части массива:
double dev(double *x,int n1,int n2){
    double x0,s=0;
    int i,n=n2-n1+1;
    x0=avr(x,n1,n2);
```

```

    for(i=n1-1;i<n2;i++) s+=x[i]*x[i];
    s/=n;
    return s-x0*x0;
}
//Отображение статистики для массива:
void ShowStat(double *x,int n){
    //Среднее:
    cout<<"Average is "<<avr(x,n)<<endl;
    //Дисперсия:
    cout<<"Dispersion is "<<dev(x,n)<<endl;
    //Стандартное отклонение:
    cout<<"Deviation is "<<sqrt(dev(x,n))<<endl;
}
//Отображение статистики для части массива:
void ShowStat(double *x,int n1,int n2){
    //Среднее:
    cout<<"Average is "<<avr(x,n1,n2)<<endl;
    //Дисперсия:
    cout<<"Dispersion is "<<dev(x,n1,n2)<<endl;
    //Стандартное отклонение:
    cout<<"Deviation is "<<sqrt(dev(x,n1,n2))<<endl;
}
//Функция для заполнения массива:
void fill(double *x,int n){
    for(int i=0;i<n;i++)
        x[i]=rand()%10;
}
//Функция для отображения значений массива:
void show(double *x,int n){
    for(int i=0;i<n;i++)
        cout<<x[i]<<" ";
    cout<<endl;
}
int main(){
    //Размер массива:
    const int N=10;
    //Массив значений:
    double z[N];
    //Заполнение массива:
    fill(z,N);
    cout<<"Base data:\n";
    //Отображение данных массива:
    show(z,N);
    cout<<"Base statistics:\n";
    //Статистика по массиву:
    ShowStat(z,N);
    cout<<"Small statistics:\n";
    //Статистика по части массива:
    ShowStat(z,3,N-2);
    return 0;}

```

У функции вычисления среднего значения `avr()` два варианта. При вычислении среднего значения по всему массиву данных аргументами функции передаются имя массива и его размер. Если для вычисления среднего значения используется только часть данных массива, вызывается вариант функции `avr()` с тремя аргументами: имя массива, порядковый номер (не индекс!) первого и последнего элементов подмассива, на основе которого вычисляется среднее. Аналогичная ситуация имеет место для функции вычисления дисперсии `dev()` и для функции отображения статистических показателей `ShowStat()`. Причем при определении функции `dev()` используется вызов соответствующего варианта функции `avr()`. В теле функции `ShowStat()` вызывается как функция `avr()`, так и функция `dev()`.

Кроме этих функций, в программе описана функция `fill()` для заполнения массива случайными числами в диапазоне от 0 до 9 включительно и функция `show()` для отображения значений массива. В результате выполнения программы получим следующее:

```
Base data:
1 7 4 0 9 4 8 8 2 4
Base statistics:
Average is 4.7
Dispersion is 9.01
Deviation is 3.00167
Small statistics:
Average is 5.5
Dispersion is 9.91667
Deviation is 3.14907
```

Для читателей, интересующихся вопросами статистики, отметим, что в данном случае вычислялась несмещенная дисперсия, или дисперсия по генеральной совокупности. Стандартное отклонение, которое также рассчитывается в программе, вычисляется как корень квадратный из дисперсии.

■ Транспонирование матрицы

В листинге 4.33 представлен программный код, с помощью которого выполняется транспонирование матриц. Для этой цели создана специальная функция. Функция является перегруженной, у нее два варианта: с одним аргументом и с двумя аргументами. Дело в том, что функция в качестве значения массив возвращать не может. Поэтому необходимо предусмотреть какой-то механизм возвращения результата (результат – транспонированная матрица).

Сами собой напрашиваются два подхода: передавать матрицу-результат аргументом функции либо вносить изменения непосредственно в исходную

матрицу. В соответствии с этим определяются и варианты функции транспонирования матриц. Исходная матрица (двумерный массив) передается первым аргументом функции `trans()`. Функция значения не возвращает. Если больше аргументов нет, изменения вносятся в эту матрицу. Если у функции есть еще один аргумент (двумерный массив такого же размера, как и первый), результат транспонирования записывается во вторую матрицу.

Листинг 4.33. Транспонирование матрицы

```
#include <iostream>
#include <cstdlib>
using namespace std;
//Размер матриц:
const N=3;
//Транспонирование матрицы (результат-второй аргумент):
void trans(double A[N][N],double B[N][N]){
    int i,j;
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            B[i][j]=A[j][i];
}
//Транспонирование матрицы (результат записывается в аргумент):
void trans(double A[N][N]){
    int i,j;
    double s;
    for(i=0;i<N;i++)
        for(j=i+1;j<N;j++){
            s=A[i][j];
            A[i][j]=A[j][i];
            A[j][i]=s;
        }
}
//Заполнение матрицы случайными числами:
void fill(double A[N][N]){
    int i,j;
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            A[i][j]=rand()%10;
}
//Вывод матрицы на экран:
void show(double A[N][N]){
    int i,j;
    for(i=0;i<N;i++){
        for(j=0;j<N;j++)
            cout<<A[i][j]<<" ";
        cout<<endl;
    }
}
int main() {
```

```

//Двумерные массивы:
double A[N][N],B[N][N];
cout<<"Initial matrix:\n";
//Заполнение массива:
fill(A);
//Отображение массива:
show(A);
cout<<"After transform:\n";
//Транспонирование:
trans(A,B);
//Результат:
show(B);
cout<<"Initial matrix:\n";
//Заполнение массива:
fill(A);
//Результат:
show(A);
cout<<"After transform:\n";
trans(A);
show(A);
return 0;
}

```

Кроме непосредственно функции для транспонирования матриц, в программе описана функция `fill()` для заполнения двумерного массива целыми случайными числами в диапазоне от 0 до 9, а также функция `show()` для вывода матрицы на экран.

В главном методе программы проверяется работа обоих вариантов перегруженной функции `trans()`. Результат выполнения программы имеет следующий вид:

```

Initial matrix:
1 7 4
0 9 4
8 8 2
After transform:
1 0 8
7 9 8
4 4 2
Initial matrix:
4 5 5
1 7 1
1 5 2
After transform:
4 1 1
5 7 5
5 1 2

```


Отметим, что при передаче двумерных массивов в качестве аргументов функций (при описании функций) размер по первому индексу можно было бы и не указывать, для второго индекса размер должен быть указан обязательно.

Резюме

Создавайте лишь немного законов, но следите за тем, чтобы они соблюдались.

Д. Локк

1. Использование функций существенно улучшает читабельность и функциональность программы. Функция – именованный блок программного кода, который может вызываться через имя не только в главном методе, но и в других функциях.
2. Функции могут передаваться аргументы (могут и не передаваться), функцией может возвращаться результат (а может и не возвращаться).
3. Результатом функции может быть любой базовый тип, указатель, ссылка, класс – практически все, кроме массива.
4. Главный метод программы – функция `main()` – также может иметь аргументы. Аргументами функции `main()` являются: количество параметров командной строки (включая имя программы) и символьный массив с названиями этих параметров.
5. Существует два механизма передачи аргументов функции: по значению и через ссылку. При передаче аргументов по значению на самом деле создается копия переменной, указанной аргументом функции. При передаче аргумента через ссылку функция имеет доступ непосредственно к переменным, указанным аргументами. По умолчанию используется механизм передачи аргументов по значению.
6. Для аргументов функции можно указывать значения по умолчанию. Если соответствующий аргумент функции явно не указан, используется значение по умолчанию. Аргументы со значением по умолчанию указываются последними в списке аргументов функции.
7. Под рекурсией подразумевают ситуацию, когда программный код функции реализуется через вызов этой же функции (как правило, с измененным аргументом). Использование рекурсии нередко упрощает структуру кода, но обычно приводит к неэффективному использованию системных ресурсов.

8. В C++ существует механизм перегрузки функций. В этом случае фактически создается несколько функций с одинаковыми названиями, но разными прототипами. Разные варианты перегруженной функции имеют разное количество или разный тип параметров, реже – разный тип результата. Механизм перегрузки используется при программировании однотипных действий (как правило, с разными данными) и позволяет для реализации общего алгоритма вызывать функцию с одним и тем же именем. Перегрузка функции является механизмом реализации концепции полиморфизма.

Контрольные вопросы

Американский народ сказал свое слово. Но чтобы понять, что он сказал, потребуется некоторое время.

В. Клинтон

1. Что такое функция? Как она объявляется?
2. Какой результат может возвращать функция?
3. Как функции передаются аргументы? Что такое передача аргумента по значению и по ссылке? Чем эти способы отличаются и как реализуются?
4. Каким образом функции в качестве значения передаются указатели?
5. Каким образом функции в качестве аргумента передаются массивы?
6. Какие аргументы у главного метода программы – функции `main()`?
7. Что такое аргументы по умолчанию и как они определяются?
8. Что такое рекурсия, и в каких случаях она используется?
9. Что такое перегрузка функции? В каких случаях применяется перегрузка?

Задачи для самостоятельного решения

Задача 1. Написать программу с функцией для вычисления числа из последовательности Фибоначчи. Аргументом функции является порядковый номер числа в последовательности.

Задача 2. Написать программу, в которой реализовать функцию для вычисления синуса $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$. Аргументом функции является переменная x , а граница ряда определяется как константа.

Задача 3. Написать программу, в которой реализовать функцию для вычисления косинуса $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$. Аргументом функции является переменная x , а граница ряда определяется как константа.

Задача 4. Написать программу, в которой реализовать функцию для вычисления экспоненты $\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$. Аргументом функции является переменная x , а граница ряда определяется как константа.

Задача 5. Написать программу, в которой реализовать функцию для вычисления гиперболического косинуса $ch(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}$. Аргументом функции является переменная x , а граница ряда определяется как константа.

Задача 6. Написать программу, в которой реализовать функцию для вычисления логарифма $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots = \sum_{n=1}^{\infty} \frac{(-1)^{n+1} x^n}{n}$. Аргументом функции является переменная x (значение $|x| < 1$), а граница ряда определяется как константа.

Задача 7. Написать программу, в которой реализовать функцию для вычисления ряда $\frac{\sin(x)}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n+1)!}$. Аргументом функции является переменная x , а граница ряда определяется как константа.

Задача 8. Написать программу, в которой реализовать функцию для вычисления двойного факториала числа $n!! = n \cdot (n-2) \cdot (n-4) \cdot \dots$. Число n является аргументом функции.

Задача 9. Написать программу для вычисления скалярного произведения двух векторов. Векторы реализуются в виде массивов, их названия передаются аргументами функции.

Задача 10. Написать программу с функцией для вычисления угла φ между двумя векторами. Воспользоваться соотношением $\vec{a} \cdot \vec{b} = |\vec{a}| \cdot |\vec{b}| \cos(\varphi)$, где через $\vec{a} \cdot \vec{b}$ обозначено скалярное произведение векторов. Аргументами функции являются названия массивов, в виде которых реализуются векторы.

Задача 11. Написать программу с функцией для вычисления определителя матрицы размерами 2×2 .

Задача 12. Написать программу с функцией для поиска экстремального (наибольшего или наименьшего) элемента массива. Массив заполнить случайными числами.

Задача 13. Написать программу с функцией для вычисления среднеквадратичного значения для элементов массива. Массив является аргументом функции. Предусмотреть возможность вычисления значения по подмножеству массива.

Задача 14. Написать программу с функцией для вычисления значения полинома. Аргументами функции являются массив коэффициентов полинома и переменная полинома, для которой вычисляется значение полинома.

Задача 15. Написать программу с функцией для вычисления значения производной от полинома. Аргументами функции являются массив коэффициентов полинома и переменная для производной от полинома, для которой вычисляется значение полинома.

Задача 16. Написать программу с функцией для решения методом Ньютона уравнения $f(x) = 0$, где функция $f(x)$ имеет вид $f(x) = \sum_{k=0}^N a_k x^k$. Коэффициенты полинома и начальная точка являются аргументами функции.

Задача 17. Написать программу с функцией для решения методом половинного деления уравнения $f(x) = 0$, где функция $f(x)$ имеет вид $f(x) = \sum_{k=0}^N a_k x^k$. Коэффициенты полинома и интервал поиска корня являются аргументами функции.

Задача 18. Написать программу с функцией для решения методом хорд уравнения $f(x) = 0$, где функция $f(x)$ имеет вид $f(x) = \sum_{k=0}^N a_k x^k$. Коэффициенты полинома и интервал поиска корня являются аргументами функции.

Задача 19. Написать программу с функциями для вычисления параметров a и b регрессионной модели вида $f(x) = ax + b$. Массивы базовых точек $\{x_i\}$ и $\{y_i\}$, по которым строится регрессия, являются аргументами функций.

Задача 20. Написать программу с функциями для вычисления параметров a и b регрессионной модели вида $f(x) = a + \frac{b}{x}$. Массивы базовых точек $\{x_i\}$ и $\{y_i\}$, по которым строится регрессия, являются аргументами функций.

Глава 5

Текстовые строки и динамические массивы

В этой главе более детально остановимся на методах работы с текстовыми строками, реализованными в виде символьных массивов. Частично этот вопрос уже освещался в главе, посвященной работе со статическими массивами, а сами символьные массивы использовались в примерах. Здесь основное внимание сосредоточим на тех особенностях, которые выделяют символьные массивы на фоне массивов прочих типов данных.

Создание и инициализация строк

Мы сейчас увидим такое, что лучше бы мы не видели его совсем.

Из к/ф «Человек с бульвара Капуцинов»

Как уже отмечалось, текстовые строки в C++ могут быть реализованы в виде символьных массивов или в виде объектов класса `string`. В этой главе речь будет идти только о первом механизме представления текстовых строк.

Объявляется символьный массив точно так же, как и любой другой массив: необходимо указать тип массива, его имя и размер (по каждому из индексов, если их несколько). Для начала остановимся на одномерных символьных массивах.

Примером объявления символьного массива может быть инструкция `char str[30]`, которой объявляется массив из 30 элементов. Далее этот массив в том или ином виде необходимо инициализировать. Обычно используют такие варианты инициализации символьных массивов: инициализация при объявлении, инициализация путем ввода значения с клавиатуры, посимвольное заполнение массива или копирование в массив строки с помощью специальных функций. Последовательно рассмотрим эти способы инициализации.

При инициализации массива одновременно с его объявлением значение текстовая строка указывается, через знак равенства, после имени массива.

Например, командой `char str[20]="Hello, World!"` инициализируется символьный массив из 20 элементов, причем в первые 13 элементов (количество символов в текстовой строке-значении) посимвольно заносится текстовая строка, 14-й элемент заполняется нуль-символом `'\0'` окончания текстовой строки, а остальные 6 элементов массива остаются временно неиспользованными. Если размер массива не указать, он будет выбран в соответствии с длиной текстовой строки с учетом наличия нуль-символа. Так, командой `char str[]="Hello, World!"` создается символьный массив из 14 элементов (13 символов в текстовой строке `Hello, World!` и еще один нуль-символ в конце). Удобство использования в качестве значения, которым инициализируется символьный массив, текстового литерала состоит в первую очередь в том, что при такой инициализации нуль-символ добавляется автоматически. Если инициализировать массив, указывая каждый символ в отдельности, в явном виде необходимо указать и нуль-символ: `char str[]={'H','e','l','l','o',' ',' ',' ','W','o','r','l','d',' ','\0'}`.

При считывании значения текстовой строки с клавиатуры с занесением этого значения в символьный массив имя массива, в который выполняется запись, указывается справа от оператора ввода. Пример приведен в листинге 5.1.

Листинг 5.1. Считывание строки с клавиатуры

```
#include <iostream>
using namespace std;
int main(){
    char str[100];
    cout<<"Enter your text, please: ";
    cin>>str;
    cout<<"Your text is: "<<str<<endl;
    return 0;
}
```

Командой `char str[100]` объявляется символьный массив из 100 элементов, а командой `cin>>str` с клавиатуры считывается текстовое значение и заносится в этот массив. Если ввести на запрос программы слово `Hello`, результат выполнения программы будет выглядеть следующим образом:

```
Enter your text, please: Hello
Your text is: Hello
```

Формально в массив можно занести текст длиной до 99 символов. Однако на практике все ограничивается одним словом. Дело в том, что оператор ввода `>>` считывает информацию, пока не встретит символ перехода

на новую строку, символ табуляции или пробел. Поскольку слова в тексте разделяются пробелами, будет считываться только первое слово. Например, если на запрос программы ввести текст `Hello, World!`, результат получим следующий:

```
Enter your text, please: Hello, World!
Your text is: Hello,
```

Проблема решается, если вместо оператора ввода `>>` использовать функцию `gets()`. Аргументом функции указывается название массива, в который заносится значение. После внесения изменений программный код будет выглядеть так, как показано в листинге 5.2.

Листинг 5.2. Считывание строки с помощью функции `gets()`

```
#include <iostream>
#include <cstdio>
using namespace std;
int main(){
    char str[100];
    cout<<"Enter your text, please: ";
    //Для считывания строки использована функция gets():
    gets(str);
    cout<<"Your text is: "<<str<<endl;
    return 0;
}
```

В общем случае для использования функции `gets()` необходимо подключить заголовок `<cstdio>`. Результат выполнения программы в этом случае может иметь следующий вид:

```
Enter your text, please: Hello, World!
Your text is: Hello, World!
```

При поэлементном заполнении символьного массива уже после объявления необходимо помнить, что последним символом после ввода символов строки должен быть нуль-символ. Пример программы с поэлементным заполнением символьного массива приведен в листинге 5.3.

Листинг 5.3. Поэлементное заполнение массива

```
#include <iostream>
#include <cstdio>
using namespace std;
int main(){
    char str[30];
    int n=26;
```



```

char s='a';
for(int i=0;i<n;i++,s++)
str[i]=s;
str[n]='\0';
cout<<str<<endl;
return 0;
}

```

Результат выполнения этой команды имеет вид:

```

abcdefghijklmnopqrstuvwxyz

```

Обращаем внимание на способ изменения символьной переменной `s`: несмотря на то, что переменная имеет тип `char`, для ее изменения используется оператор инкремента `++`. С таким же успехом вместо оператора цикла, приведенного в листинге 5.3, можно использовать следующий оператор цикла (прочие команды неизменны):

```

for(int i=0;i<n;i++)
    str[i]=s+i;

```

Если после этого заполнить массив новыми значениями, заполняется только начальная часть массива. При этом те элементы массива, которые не попали под переопределение, остаются неизменными. Обратимся к программному коду в листинге 5.4.

Листинг 5.4. Поэлементное заполнение массива

```

#include <iostream>
#include <cstdio>
using namespace std;
int main(){
    int i;
    const int n=20;
    char str[20];
    for(i=0;i<n;i++){
        str[i]='A'+i;
        cout<<str[i];
    }
    cout<<endl;
    cout<<"Enter a string: ";
    gets(str);
    cout<<str<<endl;
    for(i=0;i<n;i++)
        cout<<str[i];
    cout<<endl;
    return 0;
}

```

Сначала поэлементно заполняется символьный массив `str` размера 20 (размер задан через целочисленную константу `n`). Занесенные в массив значения также поэлементно выводятся на экран одной строкой. Перебираются все элементы массива. Для заполнения массива и вывода значений на экран используется оператор цикла. Далее пользователю предлагается ввести текстовую строку. При считывании строки заносится в массив. Строка командой `cout<<str<<endl` выводится на экран, после чего с помощью оператора цикла выводятся все элементы массива. Если в качестве строки пользователя ввести фразу `Hello, World!`, то результат выполнения программы может быть следующим:

```

ABCDEFGHIJKLMNPOQRST
Enter a string: Hello, World!
Hello, World!
Hello, World! OPQRST

```

Интерес представляют предпоследняя и последняя строки вывода результатов выполнения программы. В предпоследней строке выводится фраза `Hello, World!` (та фраза, что введена пользователем с клавиатуры). Это результат выполнения команды `cout<<str`. При считывании фразы пользователя и записи ее в массив последним после непосредственно текста строки автоматически добавляется нуль-символ. Поэтому при выполнении команды `cout<<str` вывод на экран осуществляется до этого символа. Последняя строка вывода результатов получается при выполнении команд `cout<<str[i]` в рамках последнего оператора цикла. Поскольку перебираются все элементы массива, то на экран выводятся не только элементы массива до нуль-символа, но и элементы, размещенные после этого символа. Сам нуль-символ отображается в виде пробела. В этом смысле нуль-символ является очень важным индикатором, который отделяет полезный, рабочий текст, занесенный в символьный массив, от «мусора», размещенного в хвосте массива.

Нуль-символ окончания строки

Все рано или поздно кончается.

Из к/ф «Гараж»

На использовании нуль-символа окончания строки в символьном массиве основана работа многих полезных и эффективных алгоритмов. На некоторых особенностях использования нуль-символа остановимся более подробно.

В первую очередь проиллюстрируем способ посимвольного вывода строки на экран. Соответствующий программный код приведен в листинге 5.5.

Листинг 5.5. Поэлементный вывод строки на экран

```
#include <iostream>
#include <cstdio>
using namespace std;
int main(){
    char str[20];
    cout<<"Enter a string: ";
    gets(str);
    for(int i=0;str[i];i++)
        cout<<str[i];
    cout<<endl;
    return 0;
}
```

Пользователем вводится с клавиатуры строка, а командой `gets(str)` она заносится в массив `str`. После этого в рамках оператора цикла строка выводится на экран. Обратить внимание в данном случае стоит на второй блок в инструкции `for`: в качестве проверяемого условия указано выражение `str[i]`. Здесь использовано то замечательное обстоятельство, что в C++ любое ненулевое значение интерпретируется как логическое значение `true`, а нулевое – как `false`. Поэтому пока при переборе значений не встретится нуль-символ окончания строки (который интерпретируется как логическое значение `false`), значения элементов `str[i]` интерпретируются как логические значения `true`. Возникает естественный вопрос: а что будет, если строка содержит 0? Ответ такой: все будет так же, как с любой другой строкой, поскольку число 0, занесенное в символьный массив, числом быть перестает и интерпретируется как символ. Такому «символьному» нулю соответствует логическое значение `true`.

Используя нуль-символ как признак окончания строки, можем создать простенькую функцию пользователя для вычисления реальной длины строки, занесенной в массив (сразу отметим, что в C++ для этих целей имеется встроенная функция `strlen()`). Пример программного кода приведен в листинге 5.6.

Листинг 5.6. Функция пользователя для вычисления длины строки

```
#include <iostream>
#include <cstdio>
using namespace std;
//Функция для вычисления длины строки:
int length(char *str){
    int i=0;
    while(str[i]){
        i++;
    }
}
```

```

    return i;
}
//Проверка работы функции length():
int main(){
    char str[80];
    cout<<"Enter a string: ";
    gets(str);
    cout<<"String length is "<<length(str)<<endl;
    return 0;
}

```

У функции `length()` всего один элемент – указатель на массив типа `char`. Это имя того массива, куда записана строка и длину которой следует вычислить. В теле функции `length()` инициализируется с нулевым значением целочисленная переменная `i` с нулевым значением, после чего это значение последовательно увеличивается до тех пор, пока не встретится нуль-символ окончания строки. В качестве значения возвращается переменная `i`.

В главном методе программы выводится запрос на ввод пользователем текстовой строки, строка записывается в массив, после чего вычисляется длина строки в массиве. Так, если ввести дежурную фразу `Hello, World!`, получим такой результат:

```

Enter a string: Hello, World!
String length is 13

```

С помощью нуль-символов можно создавать и более замысловатые конструкции. Например, формально в один символьный массив можно записывать одновременно несколько строк. Пример приведен в листинге 5.7.

Листинг 5.7. Запись в массив нескольких строк

```

#include <iostream>
#include <cstring>
using namespace std;
/*Функция для внесения строки в массив.
Аргументы функции – массив str1 для внесения строки,
вносимая строка str2, а также порядковый индекс n строки*/
void StringIn(char *str1,char *str2,int n){
    //Поиск позиции в массиве для записи строки
    while(n!=0){
        if(!(*str1)) n--;
        str1++;}
    //Запись строки в массив
    while(*str2){
        *str1=*str2;
        str1++;
    }
}

```

```

        str2++;
    }
    //Запись нуль-символа в конец строки в массиве
    *str1='\0';
}

/*Функция для вывода на экран строки из массива.
Аргументы функции - массив str, из которого извлекается строка,
а также порядковый индекс n извлекаемой строки*/
void StringOut(char *str,int n){
    //Поиск начала извлекаемой строки
    while(n!=0){
        if(!(*str)) n--;
        str++;}
    //Выведение строки на экран
    cout<<str<<endl;
}

//Проверка работы созданных функций
int main(){
    int i;
    //Массив для записи нескольких строк
    char str[120];
    //Массив для считывания вводимой пользователем строки
    char s[30];
    //Запись строк в массив
    for(i=0;i<3;i++){
        cout<<"Enter a string: ";
        gets(s);
        StringIn(str,s,i);}
    StringIn(str,"One more string",3);
    //Считывание строк из массива
    for(i=0;i<=3;i++)
        StringOut(str,i);
    return 0;
}

```

Идея, реализованная в приведенном программном коде, достаточно проста. Состоит она в том, что в символьный массив вносится подряд несколько строк, а в качестве разделителей используются нуль-символы. Для записи строк в массив и извлечения строк из массива в программе определяются две функции. Функция `StringIn()` используется для занесения строк в массив, а функция `StringOut()` нужна для извлечения строк из массива. Рассмотрим каждую из этих функций.

У функции `StringIn()` три аргумента: массив, в который записывается строка, непосредственно записываемая строка, а также порядковый индекс записываемой строки в массиве. Последний аргумент равен количеству

строк в массиве, которые уже туда записаны. Этот аргумент объявлен как целочисленный. Первые два аргумента функции – указатели на значения типа `char`, что соответствует именам соответствующих массивов. Если с первым аргументом такой подход понятен и оправдан, то интерпретация заносимой в массив строки в виде символьного указателя у читателя может вызвать удивление. На самом деле ничего необычного в этом нет – в C++ текстовые литералы реализуются в виде массива символов, поэтому указателю символьного типа в качестве значения можно присваивать текстовый литерал (речь об этом еще будет идти). Условно код функции `StringIn()` состоит из двух частей: сначала выполняется поиск позиции в массиве (первый аргумент функции `str1`), начиная с которой туда будет записываться строка (второй аргумент функции `str2`). После того, как место записи строки определено, выполняется посимвольное копирование строки в массив. Для определения позиции строки в массиве используется оператор цикла `while()`. Проверяемым условием (`n!=0`) является отличие от нуля значения переменной `n` – третьего аргумента функции. Непосредственно тело оператора цикла состоит из двух команд: условный оператор `if(!(*str1)) n--` позволяет уменьшить на единицу значение переменной `n`, если текущий элемент массива `str1` не является нуль-символом окончания строки. Командой `str1++` осуществляется переход к следующему элементу массива.

Таким образом, осуществляется перебор элементов массива, пока не будет найден `n`-й по счету символ окончания строки, а благодаря тому, что команда `str1++` в операторе цикла размещена после условного оператора, по завершении оператора цикла указатель `str1` будет ссылаться на первый после `n`-го нуль-символа элемент. Именно с этого символа необходимо начинать запись строки в массив. Запись осуществляется посредством еще одного оператора цикла:

```
while(*str2){
    *str1=*str2;
    str1++;
    str2++;
}
```

В этом случае цикл выполняется до тех пор, пока не будет достигнут конец записываемой в массив строки. Запись посимвольная, реализуется командой `*str1=*str2`, после чего с помощью команд `str1++` и `str2++` осуществляется переход к следующему элементу массива и следующему символу строки соответственно. После выполнения оператора цикла строка записана в массив, однако ее необходимо завершить нуль-символом, для чего после оператора цикла использована команда `*str1='\0'`.

Программный код функции `StringOut()` несколько проще. У функции два аргумента – символьный массив `str` для извлечения строки и порядковый индекс строки `n` (первая по счету строка массива имеет нулевой индекс). Как и в случае функции `StringIn()`, в функции `StringOut()` сначала выполняется поиск позиции строки в массиве (код такой же, как и при поиске позиции для записи строки в массив в функции `StringIn()`). Далее с помощью команды `cout<<str` эта строка выводится на экран. Чтобы понять, почему это происходит, следует учесть два обстоятельства. Во-первых, на момент выполнения команды `cout<<str` указатель `str` ссылается на первый символ извлекаемой строки (достигается благодаря оператору цикла в начале тела функции). Во-вторых, оператором вывода `<<` символы строки выводятся до первого нуль-символа, начиная от позиции, с которой осуществляется вывод.

В главном методе программы в рамках оператора цикла трижды выводится запрос на ввод пользователем текстовой строки. Каждый раз при считывании строки эта строка заносится в массив `s`. Далее эти строки последовательно заносятся в массив `str` с помощью команды `StringIn(str, s, i)`, где индексная переменная `i` пробегает значения от 0 до 2 включительно. После оператора цикла, как иллюстрация к тому, что вторым аргументом функции `StringIn()` может указываться текстовая строка, следует команда `StringIn(str, "One more string", 3)`. Еще один оператор цикла используется для вывода на экран строк, содержащихся в массиве `str`. Делается это с помощью вызова функции `StringOut()`. Результат выполнения программы может выглядеть, например, так (жирным шрифтом выделен текст, вводимый пользователем):

```
Enter a string: String number one
Enter a string: String number two
Enter a string: String number three
String number one
String number two
String number three
One more string
```

Отметим, что если попытаться вывести на экран с помощью оператора вывода `<<` содержимого массива, в который записано несколько строк, на экране появится только первая строка. Также стоит отметить, что использование при создании функций `StringIn()` и `StringOut()` команд вида `str1++` стало возможным благодаря тому, что соответствующие аргументы-указатели на символьные массивы (в частности, `str1`) передаются по значению.

Функции для работы со строками и символами

Будет вам и ведьма, и ведьмина вода.

Из к/ф «Чародеи»

Для работы с текстовыми строками, реализованными в виде символьных массивов, есть целый ряд встроенных функций. Одна из них уже упоминалась: это функция `strlen()` для определения длины строки, записанной в массив. В таблице 5.1 перечислены другие полезные в этом отношении функции.

Таблица 5.1. Функции для работы с текстовыми строками

Функция	Заголовок	Описание
<code>strcpy(s1, s2)</code>	<code><cstring></code>	Копирование строки <code>s2</code> в строку <code>s1</code>
<code>strcat(s1, s2)</code>	<code><cstring></code>	Строка <code>s2</code> присоединяется к строке <code>s1</code>
<code>strcmp(s1, s2)</code>	<code><cstring></code>	Сравнение строк <code>s1</code> и <code>s2</code> : если строки равны, возвращается значение 0
<code>strchr(s, ch)</code>	<code><cstring></code>	Указатель на первую позицию символа <code>ch</code> в строке <code>s</code>
<code>strstr(s1, s2)</code>	<code><cstring></code>	Указатель на первую позицию подстроки <code>s2</code> в строке <code>s1</code>
<code>atoi(s)</code>	<code><cstdlib></code>	Преобразование состоящей из цифр строки <code>s</code> в целое число типа <code>int</code>
<code>atol(s)</code>	<code><cstdlib></code>	Преобразование состоящей из цифр строки <code>s</code> в целое число типа <code>long</code>
<code>atof(s)</code>	<code><cstdlib></code>	Преобразование состоящей из цифр строки <code>s</code> в действительное число типа <code>double</code>
<code>tolower(ch)</code>	<code><cctype></code>	Преобразование буквенного символа <code>ch</code> к строчному формату
<code>toupper(ch)</code>	<code><cctype></code>	Преобразование буквенного символа <code>ch</code> к прописному формату

Понятно, что встроенных функций C++ для работы с текстом и символами намного больше, но здесь приведены те, что будут использоваться в книге. Для вызова функций необходимо подключить заголовок `<cstring>`, `<cctype>` или `<cstdlib>`, как это указано в таблице. Листинг 5.8 содержит примеры вызова функций для преобразования строк к числовым форматам.

Листинг 5.8. Функции для работы со строками и символами

```
#include <iostream>
#include <cstdio>
#include <cctype>
using namespace std;
int main(){
    int n;
    double x;
    char str1[20];
    char str2[20];
    cout<<"Enter a int-string: ";
    gets(str1);
    n=atoi(str1)/2;
    cout<<"n= "<<n<<endl;
    cout<<"Enter a double-string: ";
    gets(str2);
    x=atof(str2)+2;
    cout<<"x= "<<x<<endl;
    return 0;
}
```

Пример выполнения программы может выглядеть следующим образом (вводимые пользователем значения выделены жирным шрифтом):

```
Enter a int-string: 123
n= 61
Enter a double-string: 23.6
x= 25.6
```

Операции со строками проиллюстрированы в листинге 5.9.

Листинг 5.9. Операции со строками

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;
int main(){
    char s1[20],s2[20];
    strcpy(s1,"My name is ");
    strcpy(s2,"Alex");
    for(int i=0;s2[i];s2[i]=toupper(s2[i]),i++);
    strcat(s1,s2);
    cout<<s1<<endl;
    return 0;
}
```

Результатом выполнения этой программы является строка

```
My name is ALEX
```

Сначала с помощью функции `strcpy()` записываются значения в массивы `s1` и `s2`, после чего в строке `s2` все символы меняются на прописные (в операторе цикла использована функция `toupper()`), строка `s2` добавляется командой `strcat(s1, s2)` в конец строки `s1`, и результат выводится на экран.

Строчные литералы

*Узнать в ноже старую каретную
рессору! Какое глубокое проник-
новение в суть вещей! Впрочем,
принц всегда очень тонко анали-
зировал самые сложные ситуации.*

**Из к/ф «Приключения принца
Флоризеля»**

Как уже отмечалось, строчные литералы в C++ сохраняются в виде символического массива, причем на этот литерал автоматически создается ссылка. Это немаловажное обстоятельство существенно упрощает работу с литералами текстового типа. Особенности работы с литералами иллюстрирует листинг 5.10.

Листинг 5.10. Работа с литералами

```
#include <iostream>
using namespace std;
int main() {
    char *p, *q;
    p="Hello, World!";
    q="Hello, World!"+7;
    cout<<p<<endl;
    cout<<q<<endl;
    cout<<*p<<endl;
    p++;
    cout<<*p<<endl;
    return 0;
}
```

Результат выполнения программы выглядит так:

```
Hello, World!
World!
H
e
```

В начале программы объявляются два указателя `p` и `q` на значения символьного типа. Значение указателю `p` присваивается командой `p="Hello, World!"`, в результате чего этот указатель ссылается на первый символ соответствующего литерала – как и в случае символьного массива. Однако здесь есть одно важное исключение: указатель `p` можно изменить (команда `p++`), а вот если бы `p` был объявлен как массив (например, `char p[20]`), значение `p` менять было бы нельзя. В данном случае после выполнения команды `cout<<p` на экран выводится текстовый литерал `"Hello, World!"`. Значением `*p` является первый символ `H` указанного литерала, после команды `p++` инструкция `*p` является ссылкой на второй элемент `e` литерала.

С указателем `q` дела обстоят намного интереснее. Значение указателя определяется командой `q="Hello, World!"+7`. Инструкцией `"Hello, World!"` возвращается указатель на первый элемент этого литерала в таблице строк. Поэтому вычисление выражения `"Hello, World!"+7` осуществляется в соответствии с правилами адресной арифметики. Указателю `q` в качестве значения будет присвоен адрес 8-го символа (1-й символ плюс 7 позиций) строчного литерала, т.е. адрес символа `W`. Выполнение команды `cout<<q` приводит к отображению литерала, начиная с символа `W`. В результате на экране появляется сообщение `World!`.

Двумерные символьные массивы

Помимо одномерных символьных массивов, с успехом используются и многомерные символьные массивы. В частности, двумерный символьный массив можно рассматривать как текстовую таблицу или как одномерный массив, элементами которого являются текстовые строки (реализованные в виде символьных массивов).

Во многом работа с двумерными символьными массивами напоминает работу с двумерными массивами прочих типов. В листинге 5.11 приведен пример создания и заполнения двумерного массива.

Листинг 5.11. Объявление и заполнение двумерного массива

```
#include <iostream>
#include <cstdio>
using namespace std;
int main(){
    const int n=3;
    int i;
    char s[n][50];
    for(i=0;i<n;i++){
```

```

    cout<<"Enter a string: ";
    gets(s[i]);
}
for(i=0;i<n;i++)
    cout<<s[i]<<endl;
return 0;
}

```

Объявляется двумерный массив точно так же, как и массивы других типов: указывается тип элементов, название массива и размер по каждому индексу. В данном случае командой `char s[n][50]` (константа `const int n=3`) объявляется массив из трех элементов, каждый из которых является строкой длиной до 49 символов. Далее в рамках оператора цикла делается запрос на ввод пользователем текстовой строки, и эти строки сохраняются как элементы массива `s`. Запись строк в массив осуществляется командой вида `gets(s[i])` (индексная переменная `i` принимает значения от 0 до `n-1` включительно). В данном случае использовано то обстоятельство, что в двумерном массиве `s` инструкция вида `s[i]` является ссылкой на $(i+1)$ -ю строку (элемент с индексом `i`). На этом принципе основана и процедура вывода строк-элементов массива с помощью команды `cout<<s[i]` в рамках оператора цикла.

Результат выполнения программы может выглядеть следующим образом (жирным шрифтом выделен текст, вводимый пользователем):

```

Enter a string: First string
Enter a string: Second string
Enter a string: Third string

First string
Second string
Third string

```

Разумеется, доступ к отдельному элементу можно получить традиционным способом – указав после имени массива в квадратных скобках индексы элементов массива.

Инициализация двумерного символьного массива осуществляется так же, как и инициализация двумерных массивов других типов. Пример приведен в листинге 5.12.

Листинг 5.12. Инициализация двумерного символьного массива

```

#include <iostream>
using namespace std;
int main() {
    char s[3][100]={"Hello!","My name is Alex.","What is your name?"};

```

```
for(int i=0;i<3;i++) cout<<s[i]<<endl;
return 0;
}
```

В результате выполнения программы на экране появится серия сообщений:

```
Hello!
My name is Alex.
What is your name?
```

Надо полагать, что особых комментариев эта ситуация не требует.

Динамическое выделение памяти

Я использую не только весь ум, которым располагаю, но и весь ум, который могу взять взаймы.

Вудро Вильсон

Динамический массив от статического отличается в первую очередь тем, что на момент компиляции размер динамического массива не известен, в отличие от массивов статических, для которых размер должен быть известен уже при компиляции. Типичный пример необходимости создания динамического массива – когда размер массива определяется пользователем путем ввода числового значения с клавиатуры. Чтобы определить размер массива, необходимо сначала запустить программу, и только после этого станет известно, сколько должно быть элементов в массиве. Разумеется, можно использовать в таких ситуациях статические массивы, предусмотрев в них достаточное количество элементов. Однако такой подход грешит нерациональным использованием системных ресурсов и не всегда приемлем.

Динамические массивы реализуются посредством операторов динамического распределения памяти. В C++ для выделения области памяти используется оператор `new`, а для освобождения выделенной ранее памяти используют оператор `delete`. В частности, оператор `new` выделяет область памяти и возвращает в качестве значения указатель на первую ячейку выделенной памяти. Причем указанные операторы используются для выделения памяти как для обычных переменных, так и для массивов.

Для того чтобы динамически выделить память для переменной одного из базовых типов, объявляют указатель на значение соответствующего типа и в качестве значения этому указателю присваивают результат вызова оператора `new`. При вызове оператора `new` сразу после него указывается базовый тип переменной, для которой выделяется память. Общий синтаксис для команды выделения памяти выглядит так:

```
тип_переменной *указатель;
указатель = new тип_переменной;
```

Например, чтобы выделить память под целочисленную переменную типа `int`, достаточно воспользоваться последовательностью команд:

```
int *p;
p=new int;
```

Таким образом, значением переменной-указателя (в данном случае это `p`) является адрес ячейки, выделенной для записи значения переменной. К этой ячейке можно обращаться через указатель, как и при работе с обычными переменными.

После того, как переменная, память под которую выделялась динамически, выполнила свою задачу и потребность в ней отпала, соответствующую область памяти необходимо освободить. Формально память можно и не освобождать, но это очень дурной тон. Поэтому опускаться до такого уровня не стоит.

Очистка памяти, выделенной под переменную, осуществляется, как отмечалось, с помощью оператора `delete`. Для очистки памяти необходимо вызвать этот оператор с адресом-указателем переменной, область памяти которой освобождается. Синтаксис вызова оператора `delete` следующий:

```
delete указатель
```

Например, чтобы освободить память для ячейки с адресом `p`, используем команду

```
delete p;
```

При выделении памяти можно сразу выполнять и инициализацию. Для этого записываемое в ячейку значение указывается в круглых скобках после типа переменной, для которой выделяется память. Синтаксис выделения памяти с одновременной инициализацией следующий:

```
тип_переменной *указатель;
указатель = new тип_переменной(значение);
```

В частности, корректной является такая последовательность команд:

```
int *p;
p = new int(25);
```

В этом случае выделяется память для целочисленной переменной и в соответствующую ячейку заносится значение 25. Примеры динамического выделения памяти для переменных базовых типов приведены в листин-

ге 5.13. Здесь сразу отметим одно немаловажное обстоятельство. Дело в том, что при попытке выделения памяти может произойти ошибка, связанная с тем, что свободных ячеек просто нет. Это ошибка, которую предугадать или предотвратить практически невозможно. В таких случаях обычно предусматривают обработку исключительных ситуаций. Главное назначение такого подхода состоит в том, чтобы при возникновении соответствующей ошибки работа программы не завершилась или завершилась, но по сценарию, разработанному программистом. Если обработку исключительной ситуации не предусмотреть, работа программы при возникновении ошибки будет прервана. Хорошим тоном и признаком профессионализма является обработка исключительных ситуаций. Обработка исключительных ситуаций обсуждается в специальном приложении, размещенном на диске, прилагаемом к книге. Здесь к ней прибегать не будем, но о том, что такую обработку следует выполнять, забывать не нужно.

Листинг 5.13. Динамическое выделение памяти для переменных

```
#include <iostream>
using namespace std;
int main(){
    int *p;
    double *q;
    p=new int;
    q=new double(3.6);
    cout<<"Enter int-number: ";
    cin>>*p;
    cout<<"Address: "<<p<<endl;
    cout<<"Value: "<<*p<<endl;
    cout<<"Address: "<<q<<endl;
    cout<<"Value: "<<*q<<endl;
    delete p;
    delete q;
    return 0;
}
```

Результат выполнения программы может выглядеть следующим образом:

```
Enter int-number: 12
Address: 003225B0
Value: 12
Address: 003225B8
Value: 3.6
```

В принципе, с практической точки зрения основная особенность в работе с динамическими переменными состоит в том, что к этим переменным обращаются через указатель, а не по имени.

Динамические массивы

Управлять массивами – все равно, что управлять немногими: дело в частях и числе.

Сунь-цзы, «Трактат о военном искусстве»

Практически так же, как для обычных переменных, выделяется память для массивов. Главное отличие в синтаксисе вызова оператора `new` состоит в том, что после имени базового типа переменной указывается размер массива. При освобождении памяти, выделенной под массив, после оператора `delete` перед именем указателя на массив указывается оператор `[]`. Таким образом, для выделения памяти под массив используют синтаксис вида

```
тип_массива *указатель;
указатель = new тип_массива[размер];
```

Для освобождения памяти, выделенной под массив, используют команду вида

```
delete [] указатель;
```

Пример выделения памяти под массив приведен в листинге 5.14.

Листинг 5.14. Динамическое выделение памяти для массивов

```
#include <iostream>
using namespace std;
int main(){
    int *p,n;
    cout<<"Enter n = ";
    cin>>n;
    p=new int[n];
    for(int i=0;i<n;i++){
        p[i]=2*i+1;
        cout<<p[i]<<" ";
    }
    delete [] p;
    cout<<endl;
    return 0;
}
```

В результате выполнения программы при вводе пользователем числа 10 получим такой результат:

```
Enter n = 10
1 3 5 7 9 11 13 15 17 19
```


Суть программы проста: пользователь по запросу вводит число элементов динамического массива, после чего под этот массив выделяется память. В качестве значений элементам массива присваивается последовательность нечетных чисел, и эти значения выводятся в одну строку на экран.

Обращение к элементам динамического массива осуществляется посредством индексирования указателя на первый элемент массива. Этот механизм обращения доступен и по отношению к элементам статических массивов. В этом отношении динамические массивы не отличаются от статических.

Многомерные динамические массивы

Используя операторы динамического распределения памяти, можно создавать многомерные массивы. В данном случае рассмотрим, как создаются двумерные динамические массивы. В частности, в листинге 5.15 приведен пример программы, в которой по указанным пользователем параметрам создается двумерный массив и заполняется последовательно периодически целыми числами в диапазоне от 0 до 9 включительно.

Листинг 5.15. Создание двумерного динамического массива

```
#include <iostream>
using namespace std;
int main(){
    int **p;
    int n,m,i,j;
    cout<<"Enter 1-st size: ";
    cin>>n;
    cout<<"Enter 2-nd size: ";
    cin>>m;
    p=new int*[n];
    for(i=0;i<n;i++){
        p[i]=new int[m];
        for(j=0;j<m;j++){
            p[i][j]=(i*m+j)%10;
            cout<<p[i][j]<<" ";
        }
        cout<<endl;
    }
    for(i=0;i<n;i++)
        delete [] p[i];
    delete [] p;
    return 0;
}
```

Прежде, чем перейти непосредственно к анализу программного кода, сделаем несколько замечаний относительно принципов, в соответствии с которыми создается двумерный динамический массив. Отталкиваемся от того обстоятельства, что при создании одномерного динамического массива возвращается адрес первого его элемента и выделяется соответствующее количество ячеек памяти. Двумерный динамический массив будем создавать по тому же принципу: создаем динамические одномерные массивы строк двумерного массива, а адреса первых ячеек этих одномерных массивов занесем в еще один динамический одномерный массив. Именно этот принцип реализован в приведенном программном коде.

В программе, кроме прочего, командой `int **p` объявляется переменная `p`, которая является указателем на указатель целочисленного значения. Фактически, `p` – это и есть создаваемый двумерный динамический массив. После того, как определены размеры массива `n` и `m` (вводятся пользователем с клавиатуры), командой `p=new int*[n]` объявляется массив указателей на целые числа. В этот массив будут заноситься адреса первых ячеек строк двумерного динамического массива. Заполнение массива реализуется в рамках двойного оператора цикла

```
for (i=0; i<n; i++) {
    p[i]=new int[m];
    for (j=0; j<m; j++) {
        p[i][j]=(i*m+j)%10;
        cout<<p[i][j]<<" ";
    }
    cout<<endl;
}
```

В этом операторе (точнее, два вложенных оператора) сначала командой `p[i]=new int[m]` элементу `p[i]` массива `p` вместе с выделением места под одномерный целочисленный массив размера `m` присваивается ссылка на первый элемент этого массива. Далее с помощью второго оператора заполняются элементы вновь созданного массива. Для заполнения используется команда `p[i][j]=(i*m+j)%10`. Одновременно с этим значения элементов массива выводятся на экран.

Некоторых пояснений требует и процедура освобождения памяти, выделенной под двумерный динамический массив. Если просто воспользоваться командой `delete [] p`, будет освобождено место в памяти, занятое массивом `p` адресов первых ячеек одномерных массивов. Сами одномерные массивы при этом останутся в памяти. Поэтому перед «уничтожением» массива `p` освобождаем память, выделенную для всех одномерных массивов. Для этого используется оператор цикла

```
for (i=0; i<n; i++)
    delete [] p[i];
```

После этого освобождается место, занятое «главным» массивом адресов `p`. Для этого используется команда

```
delete[] p
```

Результат выполнения программы может выглядеть следующим образом (вводимые пользователем значения 3 и 4):

```
Enter 1-st size: 3
Enter 1-st size: 4
0 1 2 3
4 5 6 7
8 9 0 1
```

Используя тот же самый подход, с минимальными изменениями, можем создать динамический двумерный массив, содержащий в каждой из строк разное число элементов. Программный код, в котором создается такой массив, приведен в листинге 5.16.

Листинг 5.16. Создание двумерного динамического массива с разным числом элементов в строках

```
#include <iostream>
using namespace std;
int main(){
    int **p, *m;
    int n, i, j;
    cout<<"Enter rows number: ";
    cin>>n;
    m=new int[n];
    for(i=0;i<n;i++){
        cout<<"Enter size of row "<<i+1<<": ";
        cin>>m[i];}
    p=new int*[n];
    for(i=0;i<n;i++){
        p[i]=new int[m[i]];
        for(j=0;j<m[i];j++){
            p[i][j]=j+1;
            cout<<p[i][j]<<" ";}
        cout<<endl;}
    for(i=0;i<n;i++)
        delete [] p[i];
    delete [] p;
    return 0;
}
```

Алгоритм выполнения программы следующий: выводится запрос на ввод пользователем количества строк в двумерном массиве. После этого пользователь, по запросу, вводит число элементов в каждой строке массива. Как только размеры массива определены, каждая строка массива заполняется натуральными числами, начиная с 1. Значения элементов массива построчно выводятся на экран, после чего массив удаляется из памяти и программа завершает работу. Например, результат выполнения программы может быть следующим (жирным шрифтом выделены значения, вводимые пользователем):

```
Enter rows number: 4
Enter size of row 1: 5
Enter size of row 2: 3
Enter size of row 3: 4
Enter size of row 4: 2
1 2 3 4 5
1 2 3
1 2 3 4
1 2
```

Главное изменение программного кода, по сравнению с предыдущим случаем, состоит в том, что количество элементов в каждой из строк двумерного массива разное, поэтому границы изменения второго индекса двумерного массива определяются не одним целым числом, а набором чисел, которые представлены в виде элементов целочисленного массива *m*. Причем этот массив создается динамически, поскольку на момент компиляции программы общее количество строк *n* в массиве неизвестно, поэтому неизвестно и количество элементов массива *m*.

После того, как пользователь вводит значение *n* для количества строк в двумерном динамическом массиве, командой `m=new int[n]` создается динамический массив из *n* элементов для записи в него количества элементов в каждой из строк массива. В рамках оператора цикла

```
for(i=0;i<n;i++){
    cout<<"Enter size of row "<<i+1<<": ";
    cin>>m[i];}
```

пользователем вводятся значения для количества элементов в строках массива, и эти значения командой `cin>>m[i]` заносятся в динамически созданный массив *m*. Прочие изменения в коде, по сравнению с листингом 5.15, сводятся к тому, что вместо постоянного размера по второму индексу для каждой из строк необходимо указывать элементы массива *m* (для *i*-й строки это элемент `m[i]`). Кроме того, упрощен способ заполнения элементов двумерного массива (команда `p[i][j]=j+1`).

Примеры решения задач

Ну что же, народ подобрался серьезный, работа проделана большая, лично у меня сомнений нет – это дело так не пойдет.

Из к/ф «Карнавальная ночь»

В дополнение к рассмотренным в основной части примерам в этом разделе рассматривается ряд прикладных задач, в которых алгоритм решения реализуется путем создания и использования символьных и динамических массивов.

■ Раскодировщик

Создадим программу, в которой определяется функция для раскодирования текстового сообщения, представленного в виде целочисленного массива. Аргументом функции является текстовая строка (реализованная в виде символьного массива), на основе которой выполняется раскодировка, а также числовой массив, представляющий закодированный текст. Соответствующий программный код представлен в листинге 5.17.

Листинг 5.17. Раскодировщик

```
#include <iostream>
#include <cstdio>
using namespace std;
void decoder(char *s,int *cstr,int n){
    char str[100];
    int i;
    for(i=0;i<n;i++){
        str[i]=s[cstr[i]];
    }
    str[n]='\0';
    cout<<str<<endl;
}
int main(){
    char s[]="He would remember that honesty is the best policy!";
    int cstr[]={0,10,6,45,4,49,2,0,4,3,17,20,9,1,8,29,24,5};
    decoder(s,cstr,18);
    return 0;
}
```

Главный метод программы далек от идеального и предназначен исключительно для того, чтобы проиллюстрировать работу функции `decoder()`, с помощью которой выполняется раскодировка. Базовая строка задана при инициализации через символьный массив `s`. Закодированная строка пред-

ставлена в виде целочисленного массива `cstr` (значения элементов массива также задаются при инициализации массива). В результате выполнения программы отображается строка

```
Hello! How are you
```

Обращаем внимание читателя, что использованный способ кодирования сообщений позволяет задавать одни и те же символы (буквы) различными цифрами – если соответствующая буква входит в базовую строку несколько раз. Кроме того, базовая строка должна быть достаточно длинной, чтобы она содержала все буквы алфавита – иначе возникнут проблемы с кодировкой. Ситуацию можно разрешать, заменяя при кодировке те символы, которых нет в базовой строке, предопределенным набором цифр – например, элементов числового массива. В этом случае необходимо удостовериться в корректности кодирования, чтобы потом можно было выполнить обратную операцию по раскодированию. Рассмотренный пример иллюстративный, поэтому базовая строка выбрана относительно небольшой.

■ Кодировщик

Рассмотрим несколько иной способ кодирования текста, в котором в качестве базового используется двумерный символьный массив – массив текстовых строк. Кодируются не символы, как в предыдущем примере, а целые слова. Слово кодируется с помощью двух чисел: номера строки и номера слова в этой строке. Например, воспользуемся четверостишьем Омара Хайяма:

*Зачем копить добро в пустыне бытия?
Кто вечно жил средь нас? Таких не видел я!
Ведь жизнь нам в долг дана, и то на срок недолгий,
А то, что в долг дано – не собственность твоя.*

На основе этого текста составим числовой код 2 1 2 8 3 2 3 7 2 3 1 4 1 5. При этом знаки препинания мы полностью игнорируем. Что означает этот числовой код? Например, числа 2 и 1, с которых он начинается, свидетельствуют о том, что первое закодированное слово размещено во второй строке и является первым по порядку. Следующее слово (код 2 8) также размещено во второй строке и является восьмым по порядку и так далее. В результате раскодирования получаем фразу *Кто видел жизнь и жил в пустыне*.

Для выполнения описанных несложных операций по кодированию текстов составляем программный код со специальной функцией, которой и выполняется кодирование (листинг 5.18 – для удобства восприятия выполняется кодирование русского текста, способы поддержки русского текста необходимо выяснить в справке используемого читателем компилятора).

Листинг 5.18. Кодировщик

```

#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
void getWord(char *s1,char *s2,int n){
    int i=0,j=0;
    int s=1;
    while(s<n){
        if(s1[i]==' '){
            s++;
            i++;
        }
        for(;(s1[i]!='\0')&&(s1[i]!=' ')&&(s1[i]!=',')
            &&(s1[i]!='.')&&(s1[i]!='!')&&(s1[i]!='?');i++,j++)
            s2[j]=s1[i];
        s2[j]='\0';
    }
    //Подсчет слов в строке s:
    int countWords(char *s){
        int i,res=1;
        for(i=0;s[i];i++) if(s[i]==' ') res++;
        return res;
    }
    //Позиция слова s2 в строке s1:
    int findWord(char *s1,char *s2){
        int i,n;
        char s[30];
        n=countWords(s1);
        for(i=1;i<=n;i++){
            getWord(s1,s,i);
            if(strcmp(s2,s)==0) return i;
        }
        return 0;
    }
    //Кодирование строки s2 на основе текста s1:
    void encoding(char s1[4][100],char *s2,int *p,int n){
        int pos;
        char s[50];
        int i,j;
        for(i=1;i<=n/2;i++){
            getWord(s2,s,i);
            for(j=0;j<4;j++){
                pos=findWord(s1[j],s);
                if(pos){
                    p[2*i-2]=j+1;
                    p[2*i-1]=pos;
                    break;
                }
            }
        }
    }
    int main(){
        int n,i;

```

```

int *p;
char s1[4][100]={"Зачем копить добро в пустыне бытия?",
"Кто вечно жил среди нас? Таких не видел я!",
"Ведь жизнь нам в долг дана, и то на срок не долгий,",
"А то, что в долг дано- не собственность твоя!"};
char s2[50]="Кто видел жизнь и жил в пустыне?";
n=2*countWords(s2);
p=new int[n];
encoding(s1,s2,p,n);
for(i=0;i<n;i++)
cout<<p[i]<<" ";
cout<<endl;
delete [] p;
return 0;
}

```

Программа реализована посредством нескольких функций, каждая из которых предназначена для обработки текстовых строк. Некоторая трудность в решении поставленной задачи связана с тем, что необходимо обрабатывать слова. Фактически необходимо предусмотреть механизм обработки строк на уровне слов. Такая обработка подразумевает, в свою очередь, выполнение ряда базовых операций. В частности, необходимо предусмотреть возможность извлечения слова из строки, поиска в строке нужного слова и расчета количества слов в строке.

Для извлечения слова из строки предлагается функция `getWord()`. У этой функции три аргумента: 1) строка, из которой извлекается слово, 2) строка, в которую слово записывается, и 3) номер извлекаемого слова в строке.

Здесь сразу возникает проблема лингвистического характера, и связана она с тем, что подразумевать под словом. В данном случае рассматривается самый простой вариант, а именно: под словом подразумевается часть текста, разделенная пробелами.

Поиск нужного слова в строке выполняется в функции `getWord()` путем последовательного перебора пробелов в строке. Например, пятое по счету слово в строке – это слово, начинающееся после четвертого пробела. Алгоритм выполнения функции следующий. После того, как определена позиция (путем перебора пробелов), с которой начинается нужное слово, начинается посимвольное копирование символов исходной строки (первый аргумент функции) в строку, указанную вторым аргументом функции. Копирование выполняется до тех пор, пока не встретится следующий пробел, нуль-символ окончания строки или один из четырех знаков препинания (запятая, точка, знак вопроса и восклицательный знак). Последним символом во вторую строку записывается нуль-символ. Обращаем внимание читателя, что при таком определении функции для извлечения слов знаки препинания после извлекаемого слова (если такие есть) игнорируются.

Функция `countWords()` имеет всего один аргумент – строку (если точнее, то указатель на символьную переменную) и предназначена для вычисления количества слов в строке. Подсчет слов выполняется путем подсчета пробелов в текстовой строке, переданной в качестве аргумента функции.

Функция `findWord()` имеет два аргумента и в качестве результата возвращает целое число. Аргументами функции указываются исходная строка, в которой выполняется поиск слова, и строка с искомым словом. Результатом функции является позиция искомого слова в исходной строке, если это слово там есть, и ноль в случае, если слова в исходной строке нет.

Непосредственно кодирование текста выполняется в функции `encoding()`. Первым аргументом функции указывается двумерный символьный массив предопределенных размеров (четыре строки по 100 позиций в каждой с учетом нуля-символа окончания строки). На основе этого массива выполняется кодирование текста. Вторым аргументом является кодируемая строка (реализованная в виде одномерного символьного массива). Третьим аргументом функции передается указатель на целое число – имя целочисленного массива, в который заносится числовой код закодированной строки. Размер этого массива указывается четвертым аргументом функции. Отметим, что массив для записи кода будет создаваться в главном методе программы динамически, тогда же будет определяться и его размер, в соответствии с длиной шифруемой строки: количество элементов целочисленного массива с кодом в два раза больше количества слов в шифруемой строке (каждое слово кодируется двумя числами). С учетом этого обстоятельства и построен алгоритм шифровки строки.

В рамках двойного оператора цикла в шифруемой строке с помощью функции `getWord()` перебираются слова (результат извлечения слова записывается в локальный символьный массив `s`). Далее путем перебора строк базисного текста с помощью функции `findWord()` выполняется поиск нужного слова. Если слово в строке найдено, в числовой массив вносятся два значения: номер строки и номер слова в этой строке.

В главном методе программы объявляются базовый текст (двумерный символьный массив) и шифруемая строка (одномерный символьный массив) – и тот и другой текст читателю знаком. С помощью функции `countWords()` определяется количество слов в шифруемой строке (длина динамически создаваемого массива в два раза больше). Кодирование выполняется с помощью функции `encoding()`. Значения массива выводятся на экран, после чего освобождается место в памяти, занятое массивом. В результате выполнения программы получаем программный код 2 1 2 8 3 2 3 7 2 3 1 4 1 5, как и должно быть.

■ Сложение целочисленных матриц

Рассмотрим простой пример по созданию программы для вычисления суммы целочисленных квадратных матриц, в которой используется динамическое выделение памяти. Обратимся к программному коду, приведенному в листинге 5.19.

Листинг 5.19. Сложение матриц

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    //Индексные переменные и размер массивов:
    int i,j,n;
    //Двумерные массивы:
    int **A,**B,**C;
    //Ввод размера массивов (ранга матриц):
    cout<<"Enter n = ";
    cin>>n;
    //Динамическое выделение памяти под одномерные массивы:
    A=new int*[n];
    B=new int*[n];
    C=new int*[n];
    //Динамическое выделение памяти для создания двумерных массивов:
    for(i=0;i<n;i++){
        A[i]=new int[n];
        B[i]=new int[n];
        C[i]=new int[n];}
    //Заполнение массивов:
    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            A[i][j]=rand()%5;
            B[i][j]=rand()%5;
            C[i][j]=A[i][j]+B[i][j];}
    //Вывод результата (построчный). Верхние строки:
    for(i=0;i<n/2;i++){
        for(j=0;j<n;j++) cout<<A[i][j]<<" ";
        cout<<" ";
        for(j=0;j<n;j++) cout<<B[i][j]<<" ";
        cout<<" ";
        for(j=0;j<n;j++) cout<<C[i][j]<<" ";
        cout<<endl;}
    //Вывод строки со знаками операций:
    for(j=0;j<n;j++) cout<<A[n/2][j]<<" ";
    cout<<" + ";
    for(j=0;j<n;j++) cout<<B[n/2][j]<<" ";
```

```

cout<<" ";
for(j=0;j<n;j++) cout<<C[n/2][j]<<" ";
cout<<endl;
//Вывод (построчный) нижних строк результата:
for(i=n/2+1;i<n;i++){
    for(j=0;j<n;j++) cout<<A[i][j]<<" ";
    cout<<" ";
    for(j=0;j<n;j++) cout<<B[i][j]<<" ";
    cout<<" ";
    for(j=0;j<n;j++) cout<<C[i][j]<<" ";
    cout<<endl;}
return 0;
}

```

В программе создается три целочисленных двумерных динамических массива: две складываемые матрицы и матрица-результат сложения. Перед созданием матриц пользователем вводится целое число, определяющее ранг складываемых матриц (матрицы квадратные). Две исходные матрицы заполняются случайными целыми числами в диапазоне значений от 0 до 4 включительно. Элементы матрицы-результата вычисляются как сумма соответствующих элементов исходных матриц. Исходные матрицы и результат их сложения отображаются на экране. Результат выполнения программы может иметь следующий вид:

```

Enter n = 3
1 4 4 2 0 4 3 4 8
3 2 0 + 3 4 0 = 6 6 0
1 1 0 2 1 2 3 2 2

```

Данные выводятся в формате, близком к математическому представлению матриц и операции их сложения. Вывод осуществляется построчно, для чего в финальной части программы использована система вложенных операторов цикла.

■ Итерационный процесс

Ранее уже рассматривались разные способы решения уравнений. При этом основное внимание уделялось способу получения результата и непосредственно самому результату. Тем не менее, нередко кроме полученного значения для корня уравнения важно знать и те приближения для корня, что были получены на предыдущих итерациях. Здесь рассмотрим процесс поиска корня уравнения вида $x = \varphi(x)$ методом последовательных итераций с записью приближенного значения на каждом итерационном шаге в массив. Такой подход подразумевает создание динамического массива. Соответствующий программный код приведен в листинге 5.20.

Листинг 5.20. Итерационный процесс

```

#include <iostream>
#include <cmath>
using namespace std;
//Функция, определяющая уравнение:
double phi(double x){
    return sqrt(1+x);}
//Функция для отображения значений массива:
void show(double *x,int n){
    int i;
    for(i=0;i<n;i++)
        cout<<"x"<<i<<" = "<<x[i]<<endl;
    }
int main(){
    //Индексная переменная и число итераций:
    int i,n;
    //Массив приближений для корня:
    double *x;
    cout<<"Enter n = ";
    cin>>n;
    //Создание массива:
    x=new double[n+1];
    //Начальное приближение:
    cout<<"Enter x0 = ";
    cin>>x[0];
    //Вычисление корня:
    for(i=1;i<=n;i++) x[i]=phi(x[i-1]);
    //Отображение результата:
    show(x,n+1);
    return 0;
}

```

В данном случае решается уравнение $x = \sqrt{1+x}$, точное решение которого $x = \frac{1 + \sqrt{5}}{2} \approx 1.618033989$. Для решения этого уравнения в програм-

ме создается специальная функция `phi()`, которая определяет зависимость $\varphi(x) = \sqrt{1+x}$. Как уже отмечалось, итерационные приближения для корня уравнения записываются в динамический массив. Перед созданием массива пользователем вводится значение для числа итераций n , которые необходимо выполнить (нулевая итерация, т.е. начальное приближение для корня, в данном случае не учитывается). Введенное пользователем начальное значение считывается как первый элемент массива. Последующие элементы вычисляются в рамках оператора цикла на основе рекуррентно-

го соотношения. Результат вычислений отображается с помощью функции `show()`, и он может иметь следующий вид:

```
Enter n = 9
Enter x0 = 0
x0 = 0
x1 = 1
x2 = 1.41421
x3 = 1.55377
x4 = 1.59805
x5 = 1.61185
x6 = 1.61612
x7 = 1.61744
x8 = 1.61785
x9 = 1.61798
```

Обычно значения корня на разных итерациях нужны для сравнения нескольких методов решения уравнений. В этом случае разумно саму процедуру вычисления корня реализовывать в виде отдельной функции, как это было сделано, например, с функцией для отображения значений массива.

■ Вычисление косинуса

Рассмотрим программу для вычисления косинуса на основе выражения вида $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$. Аналогичные задачи уже рассматривались ранее. Здесь применим следующий подход.

1. Значение N верхней границы ряда $\sum_{n=0}^N \frac{(-1)^n x^{2n}}{(2n)!}$, которым аппроксимируется косинус, а также аргумент x вводятся пользователем с клавиатуры.
2. Члены ряда (слагаемые вида $\frac{(-1)^n x^{2n}}{(2n)!}$) заносятся в динамический массив. Заполнение этого массива реализуется в виде отдельной функции. Каждый следующий элемент вычисляется на основе предыдущего.
3. Для вывода значения косинуса создается специальная функция, которой вычисляется сумма элементов массива. У функции два аргумента: массив с членами ряда и число, определяющее количество слагаемых при суммировании.

Программный код приведен в листинге 5.21.

Листинг 5.21. Вычисление косинуса

```

#include <iostream>
#include <cmath>
using namespace std;
//Функция, для заполнения массива:
void set(double *a,int N,double x){
    int i;
    a[0]=1;
    for(i=1;i<=N;i++)
        a[i]=a[i-1]*(-1)*x*x/(2*i-1)/(2*i);
}
//Функция для отображения массива:
void show(double *a,int k){
    int i;
    for(i=0;i<=k;i++) cout<<a[i]<<" ";
    cout<<endl;}
//Функция для вычисления косинуса:
double MyCos(double *a,int k){
    double s=0;
    int i;
    for(i=0;i<=k;i++) s+=a[i];
    return s;}
int main(){
    int i,N;
    double x;
    double *a;
    //Ввод параметров:
    cout<<"Enter x = ";
    cin>>x;
    cout<<"Enter N = ";
    cin>>N;
    //Создание массива:
    a=new double[N+1];
    //Заполнение массива:
    set(a,N,x);
    //Отображение массива:
    show(a,N);
    //Разные приближения для косинуса и погрешность (в %):
    for(i=1;i<=5;i++)
        cout<<N*i/5<<" : "<<MyCos(a,N*i/5)<<" : "
        <<(1-MyCos(a,N*i/5)/cos(x))*100<<"%\n";
    //Точное (вычисленное встроенной функцией) значение:
    cout<<"cos(x) = "<<cos(x)<<endl;
    return 0;
}

```

В программе создается несколько функций. В частности, функцией `set()` заполняется массив с членами ряда. Аргументами функции указываются имя заполняемого массива, верхняя граница ряда (на единицу меньше числа элементов в массиве), а также аргумент косинуса. Именно в этой функции выполняются основные вычисления: в частности, рассчитываются члены ряда.

Функция `show()` используется для отображения элементов массива, переданного первым аргументом функции. Второй аргумент функции – верхняя граница ряда.

С помощью функции `MyCos()` вычисляется приближенное значение для косинуса. Аргументами функции передается имя массива и верхний индекс в сумме ряда. Результатом функции является сумма всех элементов массива от начального до элемента с индексом, определяемым вторым аргументом функции `MyCos()`.

В главном методе программы вводится верхняя граница ряда для вычисления косинуса, а также непосредственно аргумент косинуса. После этого создается динамический массив, и он заполняется значениями – членами ряда. Полученный массив выводится на экран.

На основе полученного ряда в разных приближениях (разное количество членов ряда) вычисляется косинус указанного пользователем аргумента. Выводится пять приближенных значений. Одновременно с вычислением косинуса вычисляется ошибка, связанная с использованием для вычисления косинуса функции пользователя вместо встроенной функции. В конце выводится значение для косинуса, рассчитанное с помощью встроенной функции. Результат выполнения программы может быть следующим (жирным шрифтом выделен ввод пользователя):

```
Enter x = 1.2
Enter N = 9
1 -0.72 0.0864 -0.0041472 0.000106642 -1.70628e-006 1.86139e-008
-1.47275e-010 8.8365e-013 -4.15835e-015
1 : 0.28 : 22.7283%
3 : 0.362253 : 0.0289643%
5 : 0.362258 : 5.09649e-006%
7 : 0.362258 : 2.42739e-010%
9 : 0.362258 : 3.33067e-014%
cos(x) = 0.362358
```

Уже первые слагаемые ряда дают основание полагать, что неплохое приближение для косинуса может быть получено с учетом даже относительно небольшого количества слагаемых. Об этом же свидетельствуют данные о погрешности вычислений.

Резюме

1. Текстовые строки в C++ реализуются с помощью объектов класса `string` или в виде символьных массивов.
2. При реализации текстовой строки в виде символьного массива этот массив (статический) должен иметь достаточную длину для посимвольной записи строки.
3. Поскольку строка (символы строки) обычно заполняет не весь массив, используется специальный символ окончания строки – нуль-символ `'\0'` (не путать с нулем!).
4. Для считывания строки с клавиатуры используются оператор ввода или специальные встроенные функции (например, функция `gets()`). Каждый способ считывания имеет свои особенности.
5. Для работы со строками, реализованными в виде массивов, используются специальные функции. Они позволяют копировать, объединять, преобразовывать, сравнивать строки и пр.
6. Строчные литералы – текст, заключенный в двойные кавычки. Строчные литералы реализуются в виде символьного массива, и на них автоматически создается ссылка.
7. Наряду с одномерными символьными массивами нередко используют двумерные символьные массивы, которые можно интерпретировать как текстовые таблицы или списки записей.
8. Динамический массив от статического принципиально отличается тем, что для динамического массива память выделяется в процессе выполнения программы, для статического – при компиляции.
9. Для динамического выделения памяти используют оператор `new`. Динамически выделенную память, после завершения использования, необходимо освободить. Делается это с помощью оператора `delete`.
10. Память динамически может выделяться как под отдельную переменную, так и под массив значений. Доступ к соответствующему фрагменту памяти осуществляется через указатель, возвращаемый при выделении памяти.

Контрольные вопросы

1. Каким образом в C++ реализуются текстовые строки?
2. Как текстовая строка реализуется в виде символьного массива? Каковы особенности такой реализации? Что такое нуль-символ и зачем он используется?
3. Что такое строчный литерал, как он создается и в чем особенности его использования?
4. Что такое динамическое выделение памяти, и в каких случаях к нему прибегают? Как динамически выделяется память под переменную?
5. Как динамически выделяется память для массива?
6. Зачем нужно освобождать неиспользуемую динамически выделенную память и как это делается?

Задачи для самостоятельного решения

В великих делах достаточно даже того, чтобы желать их сделать.

Проперций

Далее вниманию читателя предлагается список задач для самостоятельного решения, в которых подразумевается использование символьных массивов для реализации текстовых строк, а также динамических массивов, размер которых определяется в процессе выполнения программы. Представленные здесь задачи достаточно просты и предназначены для закрепления навыков работы с текстовыми строками (в виде символьных массивов) и динамических массивов.

Задача 1. Написать программу для кодирования текста на основе базовой строки (см. пример «Раскодировщик»). В программе для данных целей предусмотреть специальную функцию.

Задача 2. Написать программу для кодировки текста, в которой в качестве базовой использовать predetermined текстовую строку, а символы, которых нет в этой строке, кодировать числами из числового массива.

Задача 3. Написать программу для раскодировки текста, закодированного по принципу, описанному в предыдущем примере (т.е. на основе базовой строки и числового массива для символов, которых в этой строке нет).

Задача 4. Написать программу для раскодировки текста, закодированного по принципу, описанному в примере «Кодировщик»: каждое слово кодируется двумя цифрами: номером строки и порядковым номером слова в этой строке.

Задача 5. Написать программу с функцией для распечатки в обратном (инверсном) порядке текстовой строки, реализованной в виде символьного массива. Рассмотреть возможность использования рекурсии.

Задача 6. Написать программу для перевода строки в верхний (нижний) регистр. Использовать встроенные функции для работы со строками.

Задача 7. Написать программу для кодировки (раскодировки) текста по следующему принципу: каждый символ строки (символьный массив) преобразуется в числовой формат, к каждому числу прибавляется одна и та же постоянная величина, после чего полученные числа преобразуются в символы.

Задача 8. Написать программу для кодировки (раскодировки) текста по принципу «дополнения»: в англоязычном тексте буква а меняется на букву z, буква b меняется на букву y, буква c меняется на букву x и т.д.

Задача 9. Написать программу для умножения матрицы на число. Матрицу реализовать в виде динамического массива.

Задача 10. Написать программу для вычитания двух квадратных матриц. Матрицы реализовать в виде двумерного динамического массива.

Задача 11. Написать программу для вычисления произведения двух матриц. Матрицы реализовать в виде динамических массивов.

Задача 12. Написать программу для транспонирования матрицы, реализованной в виде двумерного динамического массива.

Задача 13. Написать программу для решения уравнения $x - \sqrt{1+x} = 0$ методом половинного деления с записью результата вычисления корня на разных итерациях в динамический массив.

Задача 14. Написать программу для решения уравнения $x - \sqrt{1+x} = 0$ методом хорд с записью результата вычисления корня на разных итерациях в динамический массив.

Задача 15. Написать программу для решения уравнения $x - \sqrt{1+x} = 0$ методом касательных (метод Ньютона) с записью результата вычисления корня на разных итерациях в динамический массив.

Задача 16. Написать программу для вычисления среднего арифметического значения по числовому динамическому массиву $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$, где x_i – элементы массива.

Задача 17. Написать программу для вычисления среднеквадратичного значения $\sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2}$ (x_i – элементы массива) по числовому динамическому массиву.

Задача 18. Написать программу для вычисления среднегармонического значения $\left(\frac{1}{n} \sum_{i=1}^n \frac{1}{x_i} \right)^{-1}$ (x_i – элементы массива) по динамическому массиву.

Задача 19. Написать программу для вычисления среднегогеометрического $\sqrt[n]{\prod_{i=1}^n x_i}$ (x_i – элементы массива) по динамическому массиву.

Задача 20. Написать программу для вычисления последовательности чисел Фибоначчи (первые два числа равны единице, а каждое следующее равно сумме двух предыдущих). Количество чисел последовательности вводится пользователем, сами числа заносятся в динамический массив.

Задача 21. Написать программу, в которой реализовать вычисление экспоненты $\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$ по следующему принципу. Количество слагаемых ряда и аргумент x вводятся пользователем. Члены ряда (слагаемые вида $\frac{x^n}{n!}$) заносятся в динамический массив.

Каждый следующий элемент массива вычисляется на основе предыдущего. Процедура заполнения массива реализуется в виде отдельной функции. Вывод значения экспоненты на экран также осуществляется с помощью специальной функции, у которой два аргумента: имя массива со слагаемыми ряда и целое число, определяющее верхнюю границу суммы (см. пример «Вычисление косинуса»).

Задача 22. Написать программу для вычисления гиперболического косинуса $ch(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}$. Аргумент x и количество

слагаемых ряда вводится пользователем с клавиатуры. Члены ряда заносятся в динамический массив. Каждый следующий элемент массива вычисляется на основе предыдущего. Процедура заполнения массива реализуется в виде отдельной функции. Вывод результата на экран также осуществляется с помощью функции (см. пример «Вычисление косинуса»).

Задача 23. Написать программу для вычисления гиперболического синуса $sh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}$. Аргумент x и количество

слагаемых ряда вводится пользователем с клавиатуры. Члены ряда заносятся в динамический массив. Каждый следующий элемент массива вычисляется на основе предыдущего. Процедура заполнения массива реализуется в виде отдельной функции. Вывод результата на экран также осуществляется с помощью функции (см. пример «Вычисление косинуса»).

Задача 24. Написать программу для вычисления ряда

$$1 + x + x^2 + x^3 + \dots = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}.$$

Аргумент x (значение $|x| < 1$) и количество слагаемых ряда вводится пользователем с клавиатуры. Члены ряда заносятся в динамический массив. Каждый следующий элемент массива вычисляется на основе предыдущего. Процедура заполнения массива реализуется в виде отдельной функции. Вывод результата на экран также осуществляется с помощью функции (см. пример «Вычисление косинуса»).

Задача 25. Написать программу для вычисления ряда

$$1 - x + x^2 - x^3 + \dots = \sum_{n=0}^{\infty} (-1)^n x^n = \frac{1}{1+x}.$$

Аргумент x (значение $|x| < 1$) и количество слагаемых ряда вводится пользователем с клавиатуры. Члены ряда заносятся в динамический массив. Каждый следующий элемент массива вычисляется на основе предыдущего. Процедура заполнения массива реализуется в виде отдельной функции. Вывод результата на экран также осуществляется с помощью функции (см. пример «Вычисление косинуса»).

Глава 6

Структуры, объединения и перечисления

Все жанры искусства хороши, кроме скучных.

Вольтер

Мощь и красота языка C++ заключается еще и в том, как эффективно этот язык позволяет оперировать и организовывать различные типы данных. В этой главе кратко остановимся на таких понятиях, как структуры, объединения и перечисления.

Структуры

Под структурами подразумевают группу переменных, объединенных общим именем. Удобство структуры состоит в первую очередь в том, что она позволяет группировать разнородные данные, что бывает весьма полезно при работе со всевозможными базами данных и записями.

Объявление структуры начинается с ключевого слова `struct`, после которого следует имя структуры и, в фигурных скобках, перечисляются поля структуры (типы и имена переменных, входящих в структуру). Общий синтаксис объявления структуры следующий:

```
struct имя{  
    тип1 поле1;  
    тип2 поле2;  
    ...  
    типN полеN;  
  
    }список_переменных;
```

После фигурных скобок, заканчивающих непосредственно описание структуры, можно указать (а можно и не указывать) список переменных структуры. Дело в том, что само по себе описание структуры эту структуру не создает. Описание структуры – это всего лишь некий шаблон, по которому впоследствии создаются переменные. Процесс создания переменных структуры можно не откладывать в долгий ящик, а создать их сразу, указав список с названиями после описания структуры.

С точки зрения использования в программе имеет смысл говорить лишь о переменных структуры. Фактически переменная структуры – это объект, который имеет имя (имя переменной структуры) и поля, тип и названия которых определяются описанием структуры. Чтобы в программе создать переменную структуры, необходимо указать имя структуры, в соответствии с описанием которой создается переменная, и имя этой переменной. Другими словами, переменная структуры в программе создается точно так же, как и переменная любого базового типа, только вместо названия типа переменной указывается название структуры.

Сама по себе переменная структуры интерес представляет больше спортивный, чем практический. Все данные, которые могут понадобиться в процессе выполнения программы, записаны в поля переменной. Обращение к полю переменной структуры осуществляется через так называемый точечный синтаксис (стандартный синтаксис для объектно-ориентированного программирования) – указывается имя переменной структуры, и через точку имя поля, к которому выполняется обращение, т.е. в формате `структура.поле`. Примеры объявления и использования структур приведены в листинге 6.1.

Листинг 6.1. Объявление и использование структуры

```
#include <iostream>
#include <cstring>
using namespace std;
struct Marks{
    char name[80];
    int phys;
    int chem;
    int maths;
}ivanov,petrov,sidorov;
struct Exams{
    double phys;
    double chem;
    double maths;
};
int main(){
    strcpy(ivanov.name,"Sergei Ivanov");
    ivanov.phys=4;
    ivanov.chem=3;
    ivanov.maths=3;
    strcpy(petrov.name,"Igor Petrov");
    petrov.phys=5;
    petrov.chem=4;
    petrov.maths=4;
```

```

strcpy(sidorov.name, "Ivan Sidorov");
sidorov.phys=5;
sidorov.chem=4;
sidorov.maths=3;
Exams LastYear, ThisYear;
LastYear.chem=4.33333;
LastYear.phys=3.66667;
LastYear.maths=3.33333;
ThisYear.chem=(double) (ivanov.chem+petrov.chem+sidorov.chem)/3;
ThisYear.phys=(double) (ivanov.phys+petrov.phys+sidorov.phys)/3;
ThisYear.maths=(double) (ivanov.maths+petrov.maths+sidorov.maths)/3;
cout<<"Last year marks:"<<endl;
cout<<"Physics "<<LastYear.phys<<endl;
cout<<"Chemistry "<<LastYear.chem<<endl;
cout<<"Mathematics "<<LastYear.maths<<endl;
cout<<endl;
cout<<"This year marks:"<<endl;
cout<<"Physics "<<ThisYear.phys<<endl;
cout<<"Chemistry "<<ThisYear.chem<<endl;
cout<<"Mathematics "<<ThisYear.maths<<endl;
return 0;
}

```

В программе объявлены две структуры: Marks и Exams. В структуру Marks входят четыре поля: символьный массив name типа char и целочисленные поля (тип int) phys, chem и maths. Одновременно с объявлением структуры выполняется объявление и переменных структуры с именами ivanov, petrov и sidorov.

Структура Marks создается в программе для хранения информации об успеваемости учащихся. Оценки по трем предметам (физика, химия и математика) заносятся в целочисленные поля phys, chem и maths соответственно. Поле-массив name предназначено для записи имени и фамилии учащегося.

Напомним, что описание структуры является лишь общим шаблоном, в соответствии с которым формируются переменные структуры. Чтобы было куда записать оценки, а также имя и фамилию, необходимо создать переменную структуры. В данном случае переменные структуры созданы путем перечисления названий в конце описания структуры.

Назначение структуры Exams состоит в хранении средних оценок по каждому из предметов в разные годы. Каждому году соответствует переменная структуры. Переменные структуры создаются непосредственно в главном

методе программы. У структуры три поля, которые имеют такие же названия (`phys`, `chem` и `maths`), однако теперь поля объявлены как имеющие тип `double`. Дело в том, что средняя оценка, в отличие от оценки учащегося по предмету, в общем случае не является целым числом.

В главном методе программы сначала выполняется инициализация полей переменных структуры `Marks`. В частности, заполнение массива `name` для переменной `ivanov` выполняется командой `strcpy(ivanov.name, "Sergei Ivanov")` (копирование строки "Sergei Ivanov" в поле-массив `name`). Обращаем внимание на способ ссылки на поле `name` переменной структуры `ivanov`: ссылка выполнена в виде `ivanov.name`. Аналогично выполняются ссылки на прочие поля переменных структур:

```
ivanov.phys=4;
ivanov.chem=3;
ivanov.maths=3;
```

Точно так же заполняются поля двух других переменных (`petrov` и `sidorov`) структуры `Marks`.

Командой `Exams LastYear, ThisYear` в главном методе программы объявляются две переменные `LastYear` и `ThisYear` структуры `Exams`. Переменная `LastYear` предназначена для записи средних оценок прошлого года. Оценки (средние) за текущий год записываются в переменную `ThisYear` структуры `Exams`. Значения полям переменной структуры `LastYear` присваиваются командами

```
LastYear.chem=4.33333;
LastYear.phys=3.66667;
LastYear.maths=3.33333;
```

Соответствующие поля переменной структуры определяются как средние арифметические по набору учащихся (их в данном случае всего три):

```
ThisYear.chem=(double) (ivanov.chem+petrov.chem+sidorov.chem) /3;
ThisYear.phys=(double) (ivanov.phys+petrov.phys+sidorov.phys) /3;
ThisYear.maths=(double) (ivanov.maths+petrov.maths+sidorov.maths) /3;
```

Инструкция `(double)` использована для явного приведения типов, поскольку по умолчанию для целочисленных операндов операция деления выполняется как деление с отбрасыванием дробной части (целочисленное деление).

Далее полученные значения для полей переменных `LastYear` и `ThisYear` структуры `Exams` выводятся на экран. Результат имеет следующий вид:


```
Last year marks:
Physics 3.66667
Chemistry 4.33333
Mathematics 3.33333
```

```
This year marks:
Physics 4.66667
Chemistry 3.66667
Mathematics 3.33333
```

По отношению к структурам можно применять операцию присваивания. Для этого две переменные должны относиться к одной структуре. В результате такого присваивания из одной переменной в другую копируются значения всех полей структуры.

Массивы структур

Преступление века на дороге не вается – его нужно организовать.

Из к/ф «Старики-разбойники»

Как и обычные переменные, структуры могут быть элементами массивов. В листинге 6.2 приведен существенно измененный и упрощенный вариант предыдущего программного кода, в котором используются массивы структур.

Листинг 6.2. Массивы структур

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
using namespace std;
struct Marks{
    char name[80];
    int phys;
    int chem;
    int maths;
};
int main(){
    const int n=3;
    bool state;
    char s[80];
    Marks students[n];
    for(int i=0;i<n;i++){
        cout<<("Student name: ");
```

```

    gets(students[i].name);
    students[i].phys=3+rand()%3;
    students[i].chem=3+rand()%3;
    students[i].maths=3+rand()%3;
}
do{
    cout<<"What is the student name? ";
    gets(s);
    if(!strcmp(s,"exit")) return 0;
    state=true;
    for(int i=0;i<n;i++){
        if(!strcmp(students[i].name,s)){
            state=false;
            cout<<"Physics: "<<students[i].phys<<endl;
            cout<<"Chemistry: "<<students[i].chem<<endl;
            cout<<"Mathematics: "<<students[i].maths<<endl;
            break;
        }
    }
    if(state) cout<<"There is no student with such name\n";
}while(true);
}

```

Пример выполнения программы может выглядеть следующим образом (жирным шрифтом выделен текст, вводимый пользователем):

```

Student name: Sergei Ivanov
Student name: Igor Petrov
Student name: Ivan Sidorov
What is the student name? Sergei Ivanov
Physics: 5
Chemistry: 5
Mathematics: 4
What is the student name? Ivan Sidorov
Physics: 3
Chemistry: 3
Mathematics: 4
What is the student name? Igor Petrovski
There is no student with such name
What is the student name? Igor Petrov
Physics: 4
Chemistry: 5
Mathematics: 4
What is the student name? exit

```

В программе командой `Marks students[n]` определяется массив `students` переменных структуры `Marks` из `n` элементов (предварительно объявлена

целочисленная константа `const int n=3`). Заполнение элементов массива выполняется в рамках оператора цикла. Индексная переменная `i` пробегает значения от 0 до `n-1` включительно. Каждый раз выводится запрос на ввод имени учащегося и это имя командой `gets(students[i].name)` записывается в поле `name` переменной структуры `students[i]` – элемента массива `students`. Оценки по трем предметам для каждого учащегося определяются как случайные числа командами `students[i].phys=3+rand()%3`, `students[i].chem=3+rand()%3` и `students[i].maths=3+rand()%3`. К базовой оценке 3 прибавляется целое случайное число в диапазоне от 0 до 2 включительно (результат команды `rand()%3` – остаток от деления случайного числа на 3).

Командой `char s[80]` объявляется символьный массив, в который будет выполняться считывание вводимого пользователем имени учащегося для отображения его оценок. Этот массив используется в операторе цикла `do{...}while(true)`. При этом стоит обратить внимание, что в качестве проверяемого условия указано логическое значение `true`, что означает, что формально цикл бесконечный. Возможность выхода из этого цикла предусмотрена в самом цикле.

В начале цикла оператором командой `cout<<"What is the student name?"` выводится запрос на ввод имени учащегося. Введенное пользователем имя с помощью команды `gets(s)` считывается и записывается в массив `s`. После того, как имя учащегося введено, выполняется условный оператор `if(!strcmp(s,"exit")) return 0`. Этой командой предусмотрена возможность не только выхода из оператора цикла, но и завершения работы программы: если пользователь в качестве имени `exit`, работа программы завершается (инструкция `return 0`).

Там же командой `state=true` логической переменной `state` (переменная используется для индикации того, найдено совпадение имен или нет) присваивается значение `true`. После этого посредством оператора цикла выполняется последовательный перебор элементов массива `students` и производится сравнение значений строк, записанных в массив `s` и в поле массива `students[i].name`. Для сравнения строк используется функция `strcmp()`. Напомним, что если строки совпадают, то в качестве значения функцией возвращается 0, поэтому в условном операторе указано условие `!strcmp(students[i].name,s)`. Если условие истинно, командой `state=false` меняется состояние логической переменной `state`, после чего выводится информация об оценках соответствующего учащегося. В конце условного оператора размещена команда `break` для преждевременного выхода из оператора цикла – если совпадение найдено, продолжать поиск не имеет смысла.

Если совпадения нет (т.е. введенное пользователем имя не найдено при просмотре полей элементов массива `students`), переменная `state` имеет значение `true`. На этот случай предусмотрена команда `if (state) cout << "There is no student with such name\n"`.

Передача структур аргументами функций

*Нужно, чтобы она тебя брала,
нужно, чтобы она тебя вела – но,
в то же время, и не уводила...*

Из к/ф «Карнавальная ночь»

Структуры могут передаваться в качестве аргументов функциям. В этом случае в качестве типа аргумента функции указывается название структуры, переменная которой будет передана в функцию. Листинг 6.3 содержит пример такой ситуации.

Листинг 6.3. Передача структур аргументами функций

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
using namespace std;
struct Marks{
    char name[80];
    int phys;
    int chem;
    int maths;
};
void set_one(Marks &str){
    cout<<("Student name: ");
    gets(str.name);
    str.phys=3+rand()%3;
    str.chem=3+rand()%3;
    str.maths=3+rand()%3;
}
void set_all(Marks *str,int m){
    for(int i=0;i<m;i++) set_one(str[i]);
}
void get(Marks *str,int m){
    bool state;
    char s[80];
    do{
```

```

    cout<<"What is the student name? ";
    gets(s);
    if(!strcmp(s,"exit")) return;
    state=true;
    for(int i=0;i<m;i++){
        if(!strcmp(str[i].name,s)){
            state=false;
            cout<<"Physiscs: "<<str[i].phys<<endl;
            cout<<"Chemistry: "<<str[i].chem<<endl;
            cout<<"Mathematics: "<<str[i].maths<<endl;
            break;
        }
    }
    if(state) cout<<"There is no student with such name\n";
}while(true);
}

int main(){
    const int n=3;
    Marks students[n];
    set_all(students,n);
    get(students,n);
    return 0;
}

```

Функциональность программы не изменилась, однако основной программный код реализован в виде нескольких функций. Функция `set_one()` предназначена для заполнения полей структуры, переданной аргументом функции. Функция объявлена как `void set_one(Marks &str)`. Аргумент функции указан как такой, что имеет тип `Marks` – тип объявленной ранее структуры. Кроме того, аргумент (переменная структуры) передается по ссылке, поэтому перед формальным именем `str` аргумента функции указывается оператор `&`. Причина передачи аргумента функции по ссылке, а не по значению (как обычно) состоит в том, что в результате выполнения функции необходимо изменить аргумент, переданный этой функции.

При выполнении кода функции выводится приглашение ввести имя учащегося и с помощью генерирования случайных чисел определяются оценки по трем предметам. Соответствующий код уже комментировался ранее.

В функции `set_all()` в рамках оператора цикла вызывается функция `set_one()` для заполнения элементов массива, указанного первым аргументом функции `set_all()`. Прототип этой функции имеет вид `void set_all(Marks *str,int m)`. Первый аргумент – указатель на переменную структуры `Marks`. В данном случае это имя заполняемого массива. Второй аргумент – размер массива. Поскольку имя массива является

ссылкой на его первый элемент, проблемы с передачей аргументов, как в предыдущем случае, не возникает – при заполнении массива будет изменяться именно тот массив, имя которого передано аргументом функции.

Функция `get()` с прототипом `void get(Marks *str, int m)` предназначена для считывания имени учащегося и вывода его оценок. Первый аргумент функции – массив, в котором выполняется поиск на предмет совпадения имен учащихся, второй аргумент – размер массива. По сравнению с соответствующим кодом в листинге 6.2, в данном случае использована инструкция `if(!strcmp(s, "exit")) return` для завершения работы функции (но не программы!). Непосредственно после инструкции `return` ничего не указывается, поскольку функция определена как такая, что не возвращает результат.

Как следствие использования описанных функций существенно упрощается код главного метода программы. Кроме инициализации константы, определяющей размер массива, и объявления массива с элементами-переменными структуры, основной код главного метода состоит из вызова двух функций (`set_all()` и `get()`) и стандартной инструкции `return 0`.

Указатели на структуры

Может быть, пока Ваше сиятельство ждет Их сиятельство, Вашему сиятельству будет угодно что-то заказать?

Из к/ф «Сильва»

При объявлении указателя на структуру, как и в случае создания указателей на значения базовых типов, указывается тип структуры, а перед именем переменной-указателя ставится оператор `*`. Этот же оператор используется для получения доступа к переменной структуры по указателю на эту переменную. Кроме того, через указатель на структуру можно обращаться непосредственно к полям структуры, для чего используют оператор `->` (стрелка, состоит из двух символов – и `>`). Например, если в программе определен указатель на переменную структуры, у которой есть поле, то доступ к этому полю можно получить с помощью инструкции `указатель->поле`. Пример использования указателей на структуры приведен в листинге 6.4.

Листинг 6.4. Указатели на структуру

```
#include <iostream>
using namespace std;
struct Numbers{
    int integer;
```

```

double real;
char symbol;
};
void show(Numbers x){
    cout<<"Integer: "<<x.integer<<endl;
    cout<<"Real: "<<x.real<<endl;
    cout<<"Symbol: "<<x.symbol<<endl;
}
int main(){
    Numbers a,b;
    Numbers *p,*q;
    p=&a;
    q=&b;
    p->integer=1;
    p->real=2.5;
    p->symbol='a';
    (*q).integer=2;
    (*q).real=5.1;
    (*q).symbol='b';
    show(a);
    show(*q);
    return 0;
}

```

В программе объявлена структура `Numbers`, имеющая три поля: типа `int`, типа `double` и типа `char`. Кроме этого, описана функция `show()`, аргументом которой является переменная структуры `Numbers`, а в результате выполнения функции выводятся значения всех полей аргумента.

В главном методе программы командой `Numbers a,b` объявляются переменные `a` и `b` структуры `Numbers`, а командой `Numbers *p,*q` объявлены указатели `p` и `q` на переменные структуры `Numbers`. Значения указателям (адреса) присваиваются командами `p=&a` и `q=&b` соответственно. При этом использован, как и в случае переменных базовых типов, оператор получения адреса `&`. Поля переменной `a` заполняются посредством команд `p->integer=1`, `p->real=2.5` и `p->symbol='a'`. В этом случае ссылка на поле реализуется через указатель `p` и оператор `->`. Поля переменной `b` заполняются более консервативным способом. Для этого использованы команды `(*q).integer=2`, `(*q).real=5.1` и `(*q).symbol='b'` соответственно. Здесь принято во внимание, что инструкция `*q` является ничем иным, как той переменной, на которую ссылается указатель `q` (т.е. переменная `b`). Аналогично при вызове функции `show()` ее аргументом можно указать как имя переменной (`a` или `b`) структуры `Numbers`, так и инструкцию вида `*p` или `*q`. В результате выполнения программы получим следующее:

```

Integer: 1
Real: 2.5
Symbol: a
Integer: 2
Real: 5.1
Symbol: b

```

Указатели на структуры находят широкое применение, и в первую очередь при составлении динамических списков. В листинге 6.5 приведен пример простой программы, в которой создается такой динамический список.

Листинг 6.5. Создание динамического списка

```

#include <iostream>
using namespace std;
//У структуры два поля: целое число и
//указатель на переменную структуры:
struct DList{
    int m;
    DList *p;
};
//Функция получения доступа к элементу динамического списка:
void getm(int k,DList *q){
    int i;
    DList *t,*t1;
    t=q;
    for(i=1;i<k;i++){
        t1=t->p;
        t=t1;}
    cout<<"Value is "<<t->m<<endl;}
int main(){
    int i,n,k;
    DList *q0,*q1,*q2;
    //Создание начального элемента списка:
    q0=new DList;
    q1=q0;
    //Определение количества элементов списка:
    cout<<"Enter n= ";
    cin>>n;
    //Создание динамического списка:
    for(i=1;i<n;i++){
        cout<<"Value m= ";
        cin>>q1->m;
        q2=new DList;
        q1->p=q2;
        q1=q2;
    }
    cout<<"Value m= ";

```



```

cin>>q1->m;
//Последний элемент списка ссылается на начальный элемент:
q1->p=q0;
//Вывод значений целочисленных полей элементов списка:
do{
    cout<<"Value for index k= ";
    cin>>k;
    //Для завершения программы нужно ввести 0:
    if(!k){
        for(i=1;i<=n;i++){
            q1=q2->p;
            delete q2;
            q2=q1;
        }
        return 0;};
    getm(k,q0);
}while(true);
}

```

В программе создается структура `DList`, у которой два поля: целочисленное поле `m` типа `int` и указатель `p` на переменную структуры типа `DList`. С помощью переменных этой структуры впоследствии реализуется динамический список – аналог целочисленного массива. В поле `m` записывается целочисленное значение, а указатель `p` служит для определения адреса следующего элемента динамического списка. Более точно, поле `p` является указателем на следующий (после текущего) элемент динамического списка.

В главном методе программы, кроме трех целочисленных переменных, командой `DList *q0,*q1,*q2` объявляются также три указателя на переменные структуры `DList`. Указатель `q0` необходим для записи адреса начального элемента динамического списка: чтобы после создания динамического списка можно было получить доступ к любому его элементу, необходимо иметь ниточку, потянув за которую распутаем весь клубок. Такой ниточкой является адрес первого элемента списка. Указатели `q1` и `q2` являются техническими, вспомогательными переменными и используются при создании списка и освобождении памяти при завершении работы программы.

Командой `q0=new DList` создается первый элемент (переменная структуры `DList`) – под него динамически выделяется память, а адрес выделенной области (первой ее ячейки) записывается в переменную-указатель `q0`. Это же значение записывается в переменную-указатель `q1` (команда `q1=q0`). Далее по запросу пользователем указывается количество элементов в динамическом списке (переменная `n`). Как только размер списка определен, запускается оператор цикла, в котором выводится приглаше-

ние для пользователя указать значение поля `m` текущего элемента списка. Введенное пользователем значение считывается и записывается в поле `m` элемента списка с помощью команды `cin>>q1->m`. При этом доступ к элементу списка осуществляется через указатель `q1`. Для заполнения поля `p` элемента списка необходимо создать новый элемент (адрес этого нового элемента является значением поля `p`). Элемент создается командой `q2=new DList`, адрес вновь созданного элемента является значением указателя `q2`. Поэтому командой `q1->p=q2` этот адрес присваивается в качестве значения полю `p` предыдущего элемента. Таким образом, указатель `q1` ссылается на предпоследний на данный момент элемент списка, а указатель `q2` ссылается на последний элемент этого списка.

После выполнения команды `q1=q2` оба указателя ссылаются на последний элемент, и цикл повторяется заново. Таких циклов будет `n-1`. С учетом того, что до запуска цикла один элемент списка был уже создан, общее количество созданных элементов равно `n`. Важно то, что в рамках каждого цикла присваиваются значения для полей предпоследнего элемента. Потому после того, как все элементы списка созданы, необходимо заполнить поля последнего элемента.

Поле `m` заполняется стандартным способом. При этом использовано то обстоятельство, что после выполнения цикла указатель `q1` ссылается на последний элемент списка. Полю `p` этого элемента присвоим в качестве значения адрес первого элемента списка, для чего используем команду `q1->p=q0`. Таким образом, мы получили не просто динамический список, а циклически связанную структуру, в которой каждый ее элемент содержит ссылку на следующий. Что касается цикличности, то это не есть обязательный атрибут динамического списка, однако такой удобный и эффектный прием позволит избавиться от многих неприятностей, главная из которых – выход за пределы списка при переборе элементов.

Во второй части главного метода программы выполняется поиск элементов в динамическом списке. Для этого используется функция `getm()`, у которой два аргумента: порядковый номер элемента в динамическом списке и адрес начального элемента списка, с которого начинается поиск. В функции используется локальная целочисленная индексная переменная и два указателя `t` и `t1` на переменные структуры `DList`.

С помощью команд `t1=t->p` и `t=t1` в операторе цикла осуществляется последовательный перебор элементов, так что в результате переменная-указатель содержит адрес `k`-го элемента динамического списка (`k` – первый аргумент функции `getm()`). Командой `cout<<"Value is "<<t->m<<endl` значение поля `m` этого элемента выводится на консоль.

Для вывода значения поля `m` нужного элемента в главном методе программы функция `getm()` вызывается с аргументами `k` и `q0`, где `k` есть вводимый пользователем с клавиатуры порядковый номер элемента списка, а `q0` – адрес первого элемента списка. Вся эта процедура реализована в виде формально бесконечного оператора цикла `do{...}while(true)`. Выход из программы предусмотрен посредством условного оператора `if(!k){...}`. Оператор выполняется, если только значение `k` равно 0. Другими словами – для завершения работы программы в качестве порядкового номера элемента необходимо ввести 0.

Однако мало завершить программу – перед завершением следует освободить память, занятую элементами динамического списка. Для этого в рамках оператора цикла выполняется блок команд:

```
q1=q2->p;
delete q2;
q2=q1;
```

В результате память, выделенная под элементы динамического списка, освобождается, после чего работа программы завершается. Результат выполнения программы может иметь такой вид (жирным шрифтом выделен ввод пользователя):

```
Enter n= 5
Value m= 1
Value m= 3
Value m= 5
Value m= 7
Value m= 9
Value for index k= 2
Value is 3
Value for index k= 4
Value is 7
Value for index k= 5
Value is 9
Value for index k= 6
Value is 1
Value for index k= 0
```

Создается список из 5 элементов. Целочисленным полям этих элементов в качестве значений присваиваются нечетные числа от 1 до 9. Далее по указанному индексу элемента определяется значение его поля. Обращаем внимание читателя, что если вводится, например, число 6 (а элементов в списке 5), выводится значение поля первого элемента – это следствие циклической замкнутости списка.

Битовые размеры поля

При описании полей структуры в явном виде можно указывать размер полей. Минимальный размер поля структуры – один бит. Общий синтаксис объявления структуры с явным указанием размеров полей имеет вид:

```
struct имя{
    тип_поля1 имя_поля1: размер1;
    тип_поля2 имя_поля2: размер2;
    ...
    тип_поляN имя_поляN: размерN;
};
```

При этом для части полей можно указывать размер, а для других – нет. Если поле имеет размер в один бит, в качестве его типа указывается `unsigned` (у числа, реализованного с помощью одного бита, не может быть знака). Пример использования структуры с явно указанным размером полей приведен в листинге 6.6.

Листинг 6.6. Явное указание размеров полей структуры

```
#include <iostream>
using namespace std;
struct BitFields{
    unsigned int state:1;
    int n:2;
    int m;
} str;
int main(){
    cout<<"Enter a number: ";
    cin>>str.m;
    str.state=str.m%2;
    str.n=str.m%4-2;
    cout<<"state = "<<str.state<<endl;
    cout<<"n ="<<str.n<<endl;
    return 0;
}
```

У структуры `BitFields` три поля: однобитовое целочисленное поле `state`, двухбитовое целочисленное поле `n` и целочисленное поле `m` (размер поля не указан).

Поле `state` может принимать всего два значения: 0 или 1. Поле `n` принимает целочисленные значения в диапазоне от -2 до 1 включительно, т.е. всего 4 возможных значения: напомним, что старший бит определяет знак числа, поэтому двухбитовое число 11 соответствует числу -2, двухбитовое число 10 соответствует числу -1, число 00 соответствует нулю и число 01 соответствует числу 1.

В главном методе программы выводится запрос на ввод пользователем целого числа. Это число записывается в поле `m` переменной `str` структуры `BitFields`. В переменную `state` заносится остаток от деления этого числа на 2, а в переменную `n` записывается остаток от деления введенного пользователем числа на 4 минус 2. Значения полей переменной `str` структуры `BitFields` выводятся на экран. В результате можем получить нечто подобное следующего:

```
Enter a number: 9
state = 1
n = -1
```

Таким образом, путем явного указания размеров полей структуры удалось добиться экономии системных ресурсов: для записи значений используется минимально необходимое количество бит. Возникает естественный вопрос: а что будет, если присваиваемое битовому полю (т.е. полю, для которого явно указан размер) значение, выходит за допустимые для этого поля пределы? Например, что будет, если полю `n` значение присваивать с помощью команды `str.n=str.m%4-4`? Ответ такой: в указанных случаях происходит автоматическое отбрасывание лишних бит. В частности, если поле `m` равно 9, а поле `n` определяется как `str.n=str.m%4-4`, то этому полю будет присвоено значение 1. Поясним это. Так, остаток от деления 9 на 4 есть 1. Если отнять 4, получим -3. В двоичном представлении число -3 имеет вид (последние три бита, все старшие биты равны 1) 101. Старшие биты отбрасываются, и в результате в двухбитовом представлении получаем 01, что соответствует числу 1.

Объединения

Под объединениями подразумевают область памяти, в которой одновременно хранится несколько различных переменных. Общий синтаксис объявления объединения подразумевает использование ключевого слова `union` и имеет следующий вид:

```
union имя_объединения{
    тип_переменной1 имя_переменной1;
    тип_переменной2 имя_переменной2;
    ...
    тип_переменнойN имя_переменнойN;
} список_экземпляров;
```

Как и в случае структуры, само по себе объявление объединения не приводит к созданию новых переменных. Поэтому необходимо создать экземпляр объединения. При объявлении экземпляра объединения в качестве типа переменной указывают имя объединения.

Ссылка на члены экземпляров объединения выполняется с помощью того же синтаксиса, что и ссылка на поля структуры: член объединения указывается через точку после имени экземпляра объединения или через оператор `->` после указателя на экземпляр объединения. Листинг 6.7 содержит примеры использования объединений.

Листинг 6.7. Использование объединений

```
#include <iostream>
using namespace std;
union nums{
    unsigned short int n;
    short int m;
};
void show(nums a){
    cout<<"n = "<<a.n<<endl;
    cout<<"m = "<<a.m<<endl;
    cout<<endl;
};
int main(){
    nums un;
    un.m=1;
    show(un);
    un.m=32767;
    show(un);
    un.m=65535;
    show(un);
    un.m=-1;
    show(un);
    un.m=-65536;
    show(un);
    return 0;
}
```

Объявленное в программе объединение `nums` содержит два члена: целочисленную беззнаковую переменную `n` типа `unsigned short int` и целочисленную переменную `m` типа `short int`. Подчеркнем, что для записи обеих переменных используется общая область памяти. Обычно объем памяти для хранения данных объединения выбирается исходя из размера самой большой (по типу) переменной. В данном случае размер памяти для хранения переменных одинаков, разница только в интерпретации записанных в эту память значений.

Ситуация следующая: при обращении к переменной `n` или `m`, являющихся членами объединения, просматривается одна и та же область памяти. Поэ-

тому изменяя переменную `n` мы изменяем переменную `m` и наоборот. На все происходящее можно посмотреть и с другой точки зрения: есть область памяти, к которой можно обращаться через разные переменные, и в зависимости от типа переменной по-разному интерпретировать записанное в память значение. Нечто подобное иллюстрирует и приведенный в листинге 6.7 программный код: в главном методе программы создается экземпляр `un` объединения `nums`, одному из членов экземпляра объединения присваивается значение, после чего проверяется значение другого члена.

Для удобства в программе описана функция `show()`, которой выводятся на экран значения членов экземпляра объединения (имя экземпляра объединения указывается аргументом функции).

Проследим, каков будет результат выполнения программного кода. Для этого необходимо вспомнить некоторые особенности форматов `unsigned short int` и `short int`. Для используемого компилятора диапазон изменения чисел типа `short int` составляет от `-32768` до `32767`. Значения для чисел типа `unsigned short int` лежат в пределах от `0` до `65535`. И в том, и в другом случае числа реализуются в виде 16-битовых двоичных последовательностей. Другими словами, в область памяти, выделенную под экземпляр объединения, записывается последовательность из 16-ти нулей и единиц. Если обращение к области памяти осуществляется через переменную `m`, то этот двоичный код интерпретируется как число типа `short int`, а при обращении к памяти через переменную `n` код интерпретируется как число типа `unsigned short int`. Но сам код один и тот же!

Сначала выполняется присваивание `m=1`. В этом случае как член `n`, так и член `m` получают одинаковые значения (точнее, одно и то же значение интерпретируется одинаково), поэтому результатом выполнения команды `show(un)` будет

```
n = 1
m = 1
```

Аналогичный результат получаем в результате присваивания `un.m=32767` (оба члена имеют одинаковое значение):

```
n = 32767
m = 32767
```

Ситуация принципиально меняется после выполнения команды `un.m=65535`. В результате получим:

```
n = 65535
m = -1
```

Чтобы понять, почему так происходит, более детально рассмотрим процедуру интерпретации чисел в двоичном коде для разных типов данных. Значе-

ние 65535 выходит за допустимые пределы диапазона данных `short int`, поэтому при записи значения в переменную `m` старшие биты в двоичном представлении числа отбрасываются. Само по себе значение 65535 в двоичном коде представляется в виде $0 \dots 00 \underbrace{111 \dots 11}_{16}$, т.е. 16-ть единиц с нулевыми старшими битами (общее количество бит зависит от специфики используемого компилятора). После отбрасывания «лишних» битов остаются последние 16 единиц.

Таким образом, в память, выделенную под экземпляр объединения, записано значение $\underbrace{111 \dots 11}_{16}$. Если это значение интерпретируется как `unsigned short int` (переменная `n`), получаем в десятичной системе значение $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + \dots + 1 \cdot 2^{15} = 2^{16} - 1 = 65535$. Если же значение интерпретируется как такое, что относится к типу `short int`, ситуация несколько иная. В этом случае старший бит отвечает за знак числа, и единичный старший бит означает, что число отрицательное. Для получения соответствующего десятичного значения необходимо выполнить побитовую инверсию, прибавить единицу, перевести число в десятичную систему счисления и добавить «минус». После побитовой инверсии получаем $\underbrace{000 \dots 00}_{16}$. Прибавив единицу, получаем число вида $\underbrace{000 \dots 01}_{16}$, что соответствует числу 1, а со знаком «минус» это -1 (значение переменной `m`).

Аналогичная ситуация складывается после выполнения команды `un.m = -1`. Описанные выше преобразования необходимо проделать в обратном порядке, после чего станет ясно, что значением переменной `n` будет 65535. Кстати, такой же результат получим, если присвоим `un.n = 65535`. А вот если выполнить команду `un.m = -65536` и затем вывести значения переменных экземпляра объединения, получим

```
n = 0
m = 0
```

Чтобы перевести число -65536 в двоичное представление, необходимо модуль этого числа (т.е. 65536) перевести в двоичный код, инвертировать каждый бит и прибавить единицу. Число 65536 в двоичном представлении имеет вид $\underbrace{000 \dots 00}_N \underbrace{1000 \dots 00}_{16}$ (в данном случае количество N старших значащих нулевых битов значения не имеет). После инвертирования получаем $\underbrace{111 \dots 11}_N \underbrace{0111 \dots 11}_{16}$. Прибавив единицу, получим $1 \underbrace{000 \dots 00}_N \underbrace{1000 \dots 00}_{16}$. В переменную `m` записываются последние 16 бит, т.е. число $\underbrace{000 \dots 00}_{16}$, или 0.

Такое же значение и у переменной `n`.

Перечисления и определение типов

Под перечислением подразумевают набор именованных целочисленных констант, которые используются для определения возможных значений переменной типа перечисления. Другими словами, перечисление определяет специальный тип данных, переменные которого могут принимать целочисленные значения, причем каждое целочисленное значение имеет имя.

Объявляется перечисление с помощью ключевого слова `enum`, после которого следует имя перечисления и список (в фигурных скобках) именованных констант:

```
enum тип_перечисления {константа1, константа2, ...,
                        константаN};
```

В принципе, после списка именованных констант можно указывать список переменных типа перечисления. Примеры использования перечислений приведены в листинге 6.8.

Листинг 6.8. Использование перечислений

```
#include <iostream>
using namespace std;
int main(){
    enum color {red, green, blue, yellow, black} car;
    color bus;
    car=green;
    bus=yellow;
    cout<<"Car is "<<car<<" and bus is "<<bus<<endl;
    return 0;
}
```

Командой `enum color{red,green,blue,yellow,black}car` создается перечисление `color`. Переменные, относящиеся к этому типу (т.е. относящиеся к перечислению `color`), могут принимать значения `red`, `green`, `blue`, `yellow` и `black`. Сразу подчеркнем, что это не текстовые значения, а числовые. Другими словами, ключевые слова `red`, `green`, `blue`, `yellow` и `black` – это названия целочисленных констант. По умолчанию первая в списке значений константа равна 0, а каждая следующая на единицу больше предыдущей. Таким образом, `red` соответствует значению 0, `green` соответствует значению 1, `blue` соответствует значению 2, `yellow` соответствует значению 3 и `black` соответствует значению 4.

В программе объявляются две переменные перечисления `color`. Переменная `car` указана при объявлении перечисления сразу после фигурной

скобки списка возможных значений. Переменная `bus` объявлена командой `color bus`. Значения этим переменным присваиваются командами `car=green` и `bus=yellow` соответственно. Тем не менее следует помнить, что переменные, на самом деле, получают целочисленные значения. В этом легко убедиться по результату выполнения программы. В частности, на экране появится сообщение

```
Car is 1 and bus is 3
```

Легко догадаться, что в данном случае вместо идентификаторов `green` и `yellow` при выводе на экран использовались соответствующие числовые значения.

Используемые по умолчанию значения для целочисленных констант в списке возможных значений могут быть изменены. В этом случае после соответствующего имени через знак равенства указывается значение соответствующей константы. В листинге 6.9 приведен несколько измененный, по сравнению с предыдущим случаем, пример такого определения.

Листинг 6.9. В списке перечисления явно указаны значения констант

```
#include <iostream>
using namespace std;
int main(){
    enum color {red=10, green, blue=100, yellow, black};
    color car,bus,van;
    car=green;
    bus=blue;
    van=black;
    cout<<"Car is "<<car<<"", bus is "<<bus<<" and van is "<<van<<endl;
    return 0;
}
```

В результате выполнения программы появится строка:

```
Car is 11, bus is 100 and van is 102
```

Константе `red` явно присвоено значение 10, поэтому константа `green` имеет значение 11 (на единицу больше). Для константы `blue` явно указано значение 100, поэтому константы `yellow` и `black` имеют значения 101 и 102 соответственно.

Достаточно удобный механизм определения новых имен для уже существующих типов реализуется с помощью ключевого слова `typedef`. Воспользовавшись объявлением вида `typedef тип новое_имя_типа`, в программе создают новое имя для типа данных. В листинге 6.10 приведен простой пример такого объявления.

Листинг 6.10. Новое имя для типа данных

```
#include <iostream>
using namespace std;
int main(){
    typedef int* IntPtr;
    int n=100;
    IntPtr p;
    p=&n;
    (*p)++;
    cout<<"n = "<<n<<endl;
    return 0;
}
```

Командой `typedef int* IntPtr` объявлено новое имя `IntPtr` для данных «указатель на целое число». Впоследствии идентификатор `IntPtr` при объявлении новых переменных, причем результат, например, команды `IntPtr p`, как если бы это было объявление `int *p`. В результате выполнения команды появится сообщение `n = 101`.

Примеры решения задач

*Излишние познания повергают
в нерешительность.*

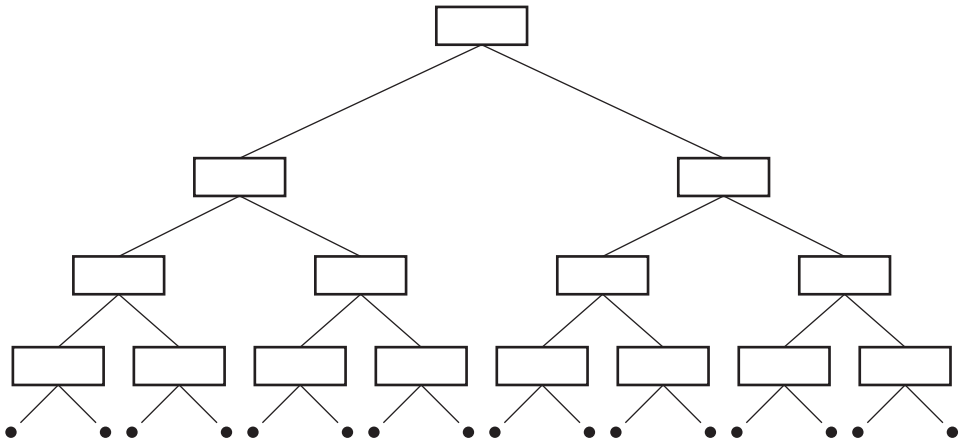
II. Бюаст

Здесь рассматриваются примеры решения некоторых задач, в которых, кроме прочего, создаются и используются в различных контекстах структуры, а также перечисления.

■ Бинарное дерево

Структуры, используемые вместе с динамическими массивами, позволяют создавать сложные структурные образования, полезные при решении самых разнообразных задач. Здесь рассмотрим создание бинарного дерева.

Под бинарным деревом в общем случае подразумевают структуру связанных данных, в которой каждый элемент указывает на два других элемента того же типа. В вершине иерархии находится один элемент (первый элемент). Он связан с двумя другими элементами (элементы второго уровня). Каждый элемент второго уровня связан с двумя элементами третьего уровня и т.д. Если бинарное дерево состоит из N уровней (начальный уровень полагается первым), то на последнем уровне количество элементов равно 2^{N-1} , а общее количество элементов дерева равняется $2^N - 1$. Схематически бинарное дерево представлено на рис. 6.1.

**Рис. 6.1. Бинарное дерево**

Для реализации такой структуры данных программными методами воспользуемся специальной функцией с рекурсивным вызовом. Весь программный код приведен в листинге 6.11.

Листинг 6.11. Создание бинарного дерева

```
#include <iostream>
using namespace std;
//Счетчик элементов дерева:
int Count=0;
//Структура-элемент дерева:
struct BinTree{
    //Поля-указатели:
    BinTree *p1;
    BinTree *p2;
    //Целочисленное поле:
    int n;
};
//Функция создания бинарного дерева:
BinTree *MakeTree(int N){
    //Указатель на создаваемый элемент:
    BinTree *p;
    p=new BinTree;
    //Создание дерева:
    Count++;
    p->n=Count;
    if(N>1){
        p->p1=MakeTree(N-1);
        p->p2=MakeTree(N-1);
    }
```

```

//Результат-указатель на созданный элемент:
return p;
}
int main(){
//Указатель на нулевой (начальный) элемент:
BinTree *q;
//Создание 4-х уровневого бинарного дерева:
q=MakeTree(4);
//Проверка результата. Количество элементов:
cout<<"Elements in tree: "<<Count<<endl;
//Элемент №1:
cout<<q->n<<endl;
//Элемент №2:
cout<<q->p1->n<<endl;
//Элемент №4:
cout<<q->p1->p1->p1->n<<endl;
//Элемент №7:
cout<<q->p1->p2->p1->n<<endl;
//Элемент №9:
cout<<q->p2->n<<endl;
//Элемент №10:
cout<<q->p2->p1->n<<endl;
//Элемент №15:
cout<<q->p2->p2->p2->n<<endl;
return 0;
}

```

При создании бинарного дерева все элементы нумеруются. Для нумерации элементов дерева вводится глобальная переменная `Count` с нулевым начальным значением. При создании каждого нового элемента дерева значение переменной увеличивается на единицу. Поэтому, например, чтобы узнать количество созданных элементов, достаточно проверить значение переменной `Count`.

Элементами бинарного дерева являются переменные структуры `BinTree`. Структура описана как такая, что имеет три поля: два указателя `p1` и `p2` на структуры того же типа и целочисленное поле `n`. Значениями указателей присваиваются адреса тех элементов-структур, на которые ссылается данный элемент. В поле `n` записывается порядковый номер элемента.

В качестве значения функция `MakeTree()` возвращает указатель на переменную структуры `BinTree`. Это именно та переменная структуры, что создается в результате вызова функции. У функции один целочисленный аргумент – количество уровней в создаваемом бинарном дереве.

Программным кодом функции реализован следующий алгоритм. Под переменную структуры типа `BinTree` динамически выделяется память, адрес

ячейки памяти присваивается локальной переменной-указателю *p*. Далее значение переменной *Count* увеличивается на единицу (начальное значение этой переменной равно нулю) и полю *n* созданной структуры присваивается значение переменной *Count*. Позднее в качестве результата функции возвращается указатель *p* на созданную структуру. После этого следует условный оператор, выполняющийся при аргументе функции, большем единицы. Таким образом, при единичном аргументе функции создается дерево с единственным элементом в иерархии. Если функция *MakeTree()* вызывается с аргументом, отличным от единицы, в рамках условного оператора кроме означенных выше операций создается еще два элемента бинарного дерева. Каждая из структур-элементов дерева создается с помощью вызова функции *MakeTree()*, но с уменьшенным на единицу аргументом. Результаты вызова этих функций присваиваются в качестве значения соответственно полям-указателям исходной структуры *p1* и *p2*.

Таким образом, в соответствии с кодом функции *MakeTree()* она рекурсивно будет вызываться до тех пор, пока аргумент в очередном вызове этой функции не станет равным единице. При этом приоритет в очередности остается за теми элементами бинарного дерева, которые связаны между собой через поля-указатели *p1* (это следствие того, что в условном операторе сначала присваивается значение полю *p1*, а затем – полю *p2*). Другими словами, сначала создается связанная цепочка элементов дерева (начиная с начального элемента), связанных по полю *p1*. Следующим создается элемент, у которого цепочка связей содержит лишь одну, реализованную через поле *p2* (на последнем уровне). Затем создается элемент, также содержащий одну *p2*-связь, но уже на предпоследнем уровне и т.д. Последним создается элемент, связанный с начальным элементом через цепочку *p2*-связей. Последовательность создания элементов бинарного дерева из четырех уровней проиллюстрирована на рис. 6.2.

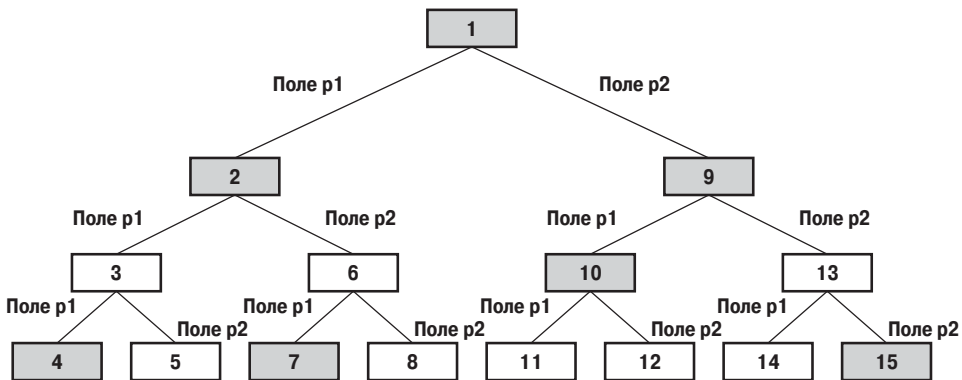


Рис. 6.2. Порядок создания элементов бинарного дерева

Цифры в квадратах соответствуют порядку создания соответствующего элемента в структуре дерева. Целочисленные поля элементов, выделенных цветом, отображаются в результате выполнения программы. В частности, в методе `main()` сначала отображается общее количество элементов в созданном дереве (переменная `Count`), а через систему ссылок отображаются поля `n` соответствующих элементов. Результат выполнения программы имеет вид:

```
Elements in tree: 15
```

```
1
2
4
7
9
10
15
```

Стоит обратить внимание на два немаловажных обстоятельства. Во-первых, поля-указатели элементов на последнем уровне остаются незаполненными. С практической точки зрения это не очень хорошо. Во-вторых, в представленной программе не предусмотрено явное освобождение памяти, занятой элементами дерева. Предлагаем читателю решить эту задачу самостоятельно.

■ Комплексные числа

Структуры можно использовать для реализации данных сложного типа. В качестве примера можно привести комплексные числа. Сразу оговоримся, что именно в этом случае использование структур не является оптимальным подходом – гораздо более эффективным представляется использование классов, к тому же в C++ есть специальный библиотечный класс для работы с комплексными числами.

Напомним, что любое комплексное число z может быть представлено в виде $z = x + iy$, где действительное число $x \equiv \operatorname{Re}(z)$ называется действительной частью числа, действительное число $y \equiv \operatorname{Im}(z)$ называется мнимой частью комплексного числа, а мнимая единица $i^2 = -1$. Приведенное представление комплексного числа называется алгебраическим. Существует также тригонометрическая форма представления комплексного числа $z = r \exp(i\varphi)$, где модуль комплексного числа r и аргумент φ связан с действительной и мнимой частями соотношениями $x = r \cos(\varphi)$ и $y = r \sin(\varphi)$.

В листинге 6.12 приведен пример программного кода, в котором для работы с комплексными числами создается специальная структура, а также

функция для вычисления результата сложения двух комплексных чисел и перевода комплексного числа из тригонометрической формы в алгебраическую.

Листинг 6.12. Комплексные числа

```
#include <iostream>
#include <cmath>
using namespace std;
//Алгебраическая форма комплексного числа:
struct ComplAlg{
    double Re;
    double Im;
};
//Тригонометрическая форма комплексного числа:
struct ComplTrig{
    double r;
    double phi;
};
//Сумма комплексных чисел (алгебраическая форма):
ComplAlg sum(ComplAlg z1, ComplAlg z2){
    ComplAlg z;
    z.Re=z1.Re+z2.Re;
    z.Im=z1.Im+z2.Im;
    return z;
}
//Перевод из тригонометрической в алгебраическую форму:
ComplAlg TrigToAlg(ComplTrig z){
    ComplAlg tmp;
    tmp.Re=z.r*cos(z.phi);
    tmp.Im=z.r*sin(z.phi);
    return tmp;}
//Отображение числа (алгебраическая форма):
void show(ComplAlg z){
    cout<<z.Re;
    if(z.Im>=0) cout<<" + "<<z.Im<<"i\n";
    else cout<<" - "<<-z.Im<<"i\n";
}
int main(){
    const double pi=3.1415;
    ComplAlg A,B,C;
    ComplTrig D;
    A.Re=1;
    A.Im=-1;
    B.Re=4;
    B.Im=5;
    D.r=2;
    D.phi=pi/6;
```



```

C=TrigToAlg(D);
show(A);
show(B);
show(C);
C=sum(A,B);
show(C);
return 0;}

```

Для реализации алгебраической и тригонометрической форм комплексного числа создаются структуры `ComplAlg` и `ComplTrig` соответственно. Сложения двух комплексных чисел, представленных в алгебраической форме, используется функция `sum()`, у которой два аргумента типа `ComplAlg`, такого же типа возвращается результат. При определении функции использовано правило, согласно которому если $z_1 = x_1 + iy_1$ и $z_2 = x_2 + iy_2$, то $z = z_1 + z_2 \equiv (x_1 + x_2) + i(y_1 + y_2)$.

Функцией `TrigToAlg()` по аргументу – переменной структуры `ComplTrig` создается переменная структуры `ComplAlg`. Функция `show()` используется для отображения комплексного числа, переданного этой функции в виде экземпляра структуры `ComplAlg`. Результат выполнения программы имеет следующий вид:

```

1 - 1i
4 + 5i
1.73207 + 0.999973i
5 + 4i

```

Аналогично можно определить функции для вычисления разности, произведения и частного для комплексных чисел, а также функцию перевода чисел из алгебраической формы в тригонометрическую.

■ Комплексная экспонента

Рассмотренный выше пример можно взять за основу для расширения общеизвестных функций на множество комплексных чисел. В частности, можем определить функцию, которая будет вычислять значение $\exp(z)$ для комплексного аргумента $z = x + iy$. Напомним, что по определению $\exp(z) = \exp(x)(\cos(y) + i\sin(y))$, т.е. $\operatorname{Re}(\exp(z)) = \exp(x)\cos(y)$ а $\operatorname{Im}(\exp(z)) = \exp(x)\sin(y)$. Воспользуемся этими соотношениями для создания соответствующей функции (листинг 6.13).

Листинг 6.13. Комплексная экспонента

```

#include <iostream>
#include <cmath>
using namespace std;

```

```
//Алгебраическая форма комплексного числа:
struct Compl{
    double Re;
    double Im;
};

//Комплексная экспонента:
Compl ComplExp(Compl z){
    Compl tmp;
    tmp.Re=exp(z.Re)*cos(z.Im);
    tmp.Im=exp(z.Re)*sin(z.Im);
    return tmp;
}

//Отображение комплексного числа:
void show(Compl z){
    cout<<z.Re;
    if(z.Im>=0) cout<<" + "<<z.Im<<"i\n";
    else cout<<" - "<<-z.Im<<"i\n";
}

int main(){
    Compl A,B;
    A.Re=1;
    A.Im=-2;
    B=ComplExp(A);
    show(A);
    show(B);
    return 0;}
```

В результате выполнения программы получим:

```
1 - 2i
-1.1312 - 2.47173i
```

Хочется верить, что представленный выше программный код особых комментариев не требует.

■ Расстояние между точками

Как известно, если две точки A и B имеют соответственно координаты $A = (x_1, y_1, z_1)$ и $B = (x_2, y_2, z_2)$, то расстояние между этими точками вычисляется как $|AB| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$. Рассмотрим программу, в которой точки реализуются в виде экземпляров структур, которые имеют по три поля типа `double` – координаты точки. Расстояние между точками вычисляется с помощью специальной функции. Соответствующий программный код приведен в листинге 6.14.

Листинг 6.14. Расстояние между точками

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point{
    double x;
    double y;
    double z;
};
double dist(Point A,Point B){
    double s;
    s=sqrt(pow(A.x-B.x,2)+ pow(A.y-B.y,2)+ pow(A.z-B.z,2));
    return s;}
int main(){
    Point A,B;
    A.x=1; A.y=1; A.z=1;
    B.x=3; B.y=2; B.z=-1;
    cout<<"Distance is "<<dist(A,B)<<endl;
    return 0;}
```

В результате выполнения этой программы получаем вполне ожидаемое сообщение

```
Distance is 3
```

Это результат расчета расстояния между точками $A = (1,1,1)$ и $B = (3,2,-1)$.

■ Пересечение прямых

Практически так же реализуется программа, в которой вычисляется точка пересечения прямых на плоскости. Каждая прямая на плоскости может быть представлена уравнением вида $ax + by + c = 0$, где x и y – декартовы координаты, а a , b и c – числовые параметры. Нетривиальной задачей делает то обстоятельство, что прямые могут пересекаться, а могут и не пересекаться (прямые параллельны). Кроме того, теоретически возможна ситуация, когда прямые совпадают. Программный код с функцией, в которой отслеживаются все эти ситуации, приведен в листинге 6.15.

Листинг 6.15. Пересечение прямых

```
#include <iostream>
#include <cmath>
using namespace std;
struct Line{
    double a;
```

```

double b;
double c;
};
void cross(Line A, Line B) {
    double x0, y0;
    if (A.a*B.b == A.b*B.a) {
        if (A.a*B.c != A.c*B.a) cout << "Parallel lines!\n";
        else cout << "Lines coincide!\n";
    }
    else {
        x0 = (A.b*B.c - B.b*A.c) / (A.a*B.b - A.b*B.a);
        y0 = (A.a*B.c - A.c*B.a) / (A.b*B.a - B.b*A.a);
        cout << "x0 = " << x0 << " y0 = " << y0 << endl;
    }
}
int main() {
    Line L1, L2, L3, L4;
    L1.a = 2; L1.b = -1; L1.c = 0;
    L2.a = 6; L2.b = -3; L2.c = 6;
    L3.a = 10; L3.b = -5; L3.c = 0;
    L4.a = 2; L4.b = 0; L4.c = -8;
    cross(L1, L2);
    cross(L1, L3);
    cross(L1, L4);
    return 0;
}

```

В общем случае координаты точки пересечения прямых, заданных уравнениями $a_1x + b_1y + c_1 = 0$ и $a_2x + b_2y + c_2 = 0$, определяются как

$x_0 = \frac{b_1c_2 - b_2c_1}{b_2a_1 - b_1a_2}$ и $y_0 = \frac{a_1c_2 - a_2c_1}{a_2b_1 - a_1b_2}$, однако эти соотношения справедли-

вы, если $b_2a_1 \neq b_1a_2$. Если $b_2a_1 = b_1a_2$ и при этом $b_1c_2 \neq b_2c_1$ (то же, что и $a_1c_2 \neq a_2c_1$), прямые не пересекаются – они параллельны. При одновременном выполнении условий $b_2a_1 = b_1a_2$ и $b_1c_2 = b_2c_1$ (это автоматически означает, что $a_1c_2 = a_2c_1$) прямые совпадают. Именно эти соотношения использовались при определении точки пересечения прямых.

В основном методе программы создается четыре экземпляра структуры Line, после чего полям присваиваются значения и вызывается функция cross(). Результат выполнения программы имеет следующий вид:

```

Parallel lines!
Lines coincide!
x0 = 4 y0 = 8

```

В данном случае рассматривались прямые, заданные уравнениями $2x - y = 0$, $6x - 3y + 6 = 0$, $10x - 5y = 0$ и $2x - 8 = 0$. Вторая прямая параллельна первой, третья и первая прямые совпадают, а четвертая прямая пересекает первую в точке с координатами $x_0 = 4$ и $y_0 = 8$.

■ Корни квадратного уравнения

Похожим образом может быть решена задача о вычислении корней квадратного уравнения. Любое квадратное уравнение вида $ax^2 + bx + c = 0$ однозначно определяется набором из трех параметров a , b и c . Для хранения этих параметров в одном объекте можно создать структуру. В листинге 6.16 приведен пример программы, в которой используется специальная структура для операций с квадратичными полиномами, и в частности для вычисления корней такого полинома. Корень вычисляется с помощью специальной функции.

Листинг 6.16. Корни квадратного уравнения

```
#include <iostream>
#include <cmath>
using namespace std;
struct Polynom{
    double a;
    double b;
    double c;
};
void roots(Polynom p){
    double x1,x2,D;
    D=p.b*p.b-4*p.a*p.c;
    if(p.a==0){
        if(p.b!=0) cout<<"x = "<<p.c/p.b<<endl;
        else if(p.c==0) cout<<"x is any number\n";
        else cout<<"There are no roots!\n";
    }
    else{
        if(D==0) cout<<"The only root is "<<-p.b/2/p.a<<endl;
        else if(D>0){
            x1=(-p.b+sqrt(D))/2/p.a;
            x2=(-p.b-sqrt(D))/2/p.a;
            cout<<"Real roots are:\n";
            cout<<"x1 = "<<x1<<" x2 = "<<x2<<endl;
        }
        else{
            cout<<"Complex roots are:\n";
            cout<<"x1 = "<<-p.b/2/p.a<<"+"<<sqrt(-D)/2/p.a<<"i\n";
            cout<<"x2 = "<<-p.b/2/p.a<<-sqrt(-D)/2/p.a<<"i\n";
        }
    }
}

int main(){
    Polynom P1,P2,P3,P4,P5,P6;
```

```

P1.a=1; P1.b=-5; P1.c=6;
P2.a=0; P2.b=2; P2.c=3;
P3.a=0; P3.b=0; P3.c=3;
P4.a=0; P4.b=0; P4.c=0;
P5.a=4; P5.b=4; P5.c=1;
P6.a=1; P6.b=1; P6.c=1;
roots(P1);
roots(P2);
roots(P3);
roots(P4);
roots(P5);
roots(P6);
return 0;}

```

В программе в функции `roots()` при вычислении корней уравнения рассмотрены все возможные случаи, в том числе когда все три коэффициента уравнения равны нулю (в этом случае решение – любое число). Также вычисляются комплексные корни. Результат выполнения программы имеет вид:

```

Real roots are:
x1 = 3 x2 = 2
x = 1.5
There are no roots!
x is any number
The only root is -0.5
Complex roots are:
x1 = -0.5+0.866025i
x2 = -0.5+0.866025i

```

В данном случае в главном методе программы с помощью предложенной функции `roots()` последовательно решались уравнения $x^2 - 5x + 6 = 0$, $2x + 3 = 0$, $0 \cdot x + 3 = 0$, $0 \cdot x^2 + 0 \cdot x + 0 = 0$, $4x^2 + 4x + 1 = 0$ и $x^2 + x + 1 = 0$.

Резюме

Неумеренная жажда знания очень часто доводит до совершенной тупости.

М. Монтень

1. Структура представляет собой набор переменных, объединенных общим именем.
2. Описание структуры начинается с ключевого слова `struct`, после которого следует имя структуры и в фигурных скобках перечисляются поля структуры. Поля указываются вместе с идентификаторами типов.

3. Описание структуры представляет собой общий шаблон, на основании которого создаются переменные (или экземпляры) структуры. Экземпляр (переменная) структуры объявляется так же, как и обычная переменная, только вместо типа переменной указывается имя структуры.
4. Обращение к полям переменной структуры осуществляется с помощью «точечного синтаксиса»: указывается имя переменной структуры и через точку имя поля.
5. Структуры могут объединяться в массивы. При создании массива структур в качестве типа элементов массива указывается имя структуры.
6. Структуры могут передаваться в качестве аргументов функциям. В качестве типа аргумента указывается имя структуры.
7. Указатели на структуры создаются так же, как и указатели на переменные базовых типов: в качестве типа указывается имя структуры, при объявлении указателей используют оператор `*`, а адрес структуры получают с помощью оператора `&`.
8. С помощью указателя на структуру можно получать доступ к полям структуры. Для этого используют оператор «стрелка» (оператор `->`): после указателя указывается оператор `->` и имя поля, к которому выполняется обращение. Можно также использовать оператор `*`: оператор указывается перед указателем на структуру, а после этой конструкции через точку указывается имя нужного поля.
9. При объявлении структур для ее полей можно в явном виде указывать размер выделяемой памяти.
10. Объединением называется область памяти, в которой хранится сразу несколько переменных. Объявление объединений выполняется с помощью ключевого слова `union`.
11. Под перечислением подразумевают специальный тип данных, переменные которого могут принимать целочисленные значения, причем каждое целочисленное значение имеет имя.
12. Объявляется перечисление с помощью ключевого слова `enum`, после которого следует имя перечисления и, в фигурных скобках, список именованных констант для определения возможных значений переменной перечисления.
13. Для уже существующих типов данных могут определяться новые имена. Делается это с помощью ключевого слова `typedef`.

Контрольные вопросы

*То, что мы знаем, – ограничено,
а то, чего мы не знаем, – бесконечно.*

II. Лаплас

1. Что такое структура, как она определяется?
2. Что такое поля структуры и как к ним выполняется обращение?
3. Чем переменная (экземпляр) структуры отличается от непосредственно структуры?
4. Каким образом создаются массивы структур?
5. Как создаются указатели на экземпляры структуры? Каким образом через указатель осуществляется обращение к полям структуры?
6. Каким образом в явном виде определяются размеры полей структуры?
7. Что такое объединение и чем объединение отличается от структуры?
8. Что такое перечисление и как оно создается?
9. Каким образом в программе для уже существующих типов данных определяются новые названия?

Задачи для самостоятельного решения

Далее представлен краткий список задач для самостоятельного решения. Основная часть задач ориентирована на создание структур. Кроме этого, в некоторых задачах необходимо создавать динамические массивы структур.

Задача 1. Написать программу для создания динамической конструкции из элементов структуры, каждый из которых ссылается на два других элемента. Ссылка осуществляется через поля-указатели. Элементы организуются по следующему принципу. Начальный (первый) элемент ссылается на два элемента (второй и третий), каждый из которых ссылается на четвертый общий элемент и друг на друга. Четвертый элемент ссылается на два элемента и т.д. «Элементарная ячейка» такой конструкции представлена на рис. 6.3.

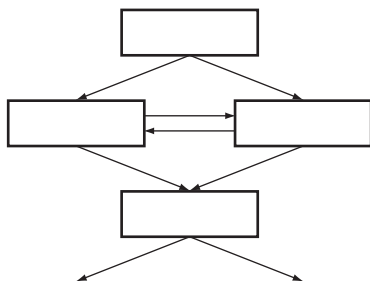


Рис. 6.3. Ячейка динамической структуры из четырех однотипных элементов

Задача 2. Написать программу для создания динамической конструкции из элементов структуры, каждый из которых ссылается на два других элемента. Ссылка осуществляется через поля-указатели. Элементы организуются по следующему принципу. Начальный (первый) элемент ссылается на два элемента (второй и третий), каждый из которых ссылается друг на друга и на еще один элемент (четвертый и пятый). Четвертый и пятый элементы ссылаются друг на друга и на один общий элемент (шестой), который ссылается на два элемента, и т.д. «Элементарная ячейка» такой конструкции представлена на рис. 6.4.

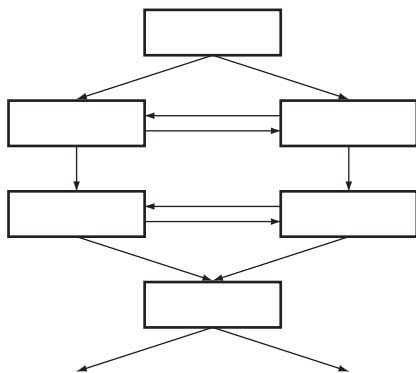


Рис. 6.4. Ячейка динамической структуры из шести однотипных элементов

Задача 3. Написать программу для создания динамической конструкции из элементов структуры двух типов («Двойной список»). Элементы первого типа содержат поля-указатели для ссылок на один элемент, а элементы второго типа имеют поля-указатели на два элемента. Элементы организуются по следующему принципу. Начальными в конструкции являются два элемента первого типа (первый и второй). Каждый из них ссылается на эле-

мент второго типа (третий и четвертый). Эти элементы ссылаются друг на друга и на элементы первого типа (пятый и шестой) и т.д. Две идущие подряд «элементарные ячейки» такой конструкции представлены на рис. 6.5. Элементы разного типа выделены разным цветом.

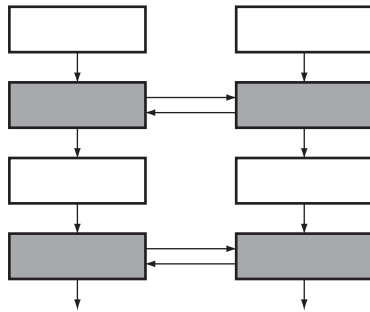


Рис. 6.5. «Двойной список»

Задача 4. Написать программу для создания динамической конструкции из элементов структуры двух типов. Элементы первого типа имеют поля-указатели для ссылки на два элемента, а элементы второго типа могут ссылаться только на один элемент. Элементы организуются по следующему принципу. Начальный (первый) элемент первого типа ссылается на два элемента второго типа (второй и третий), каждый из которых ссылается на четвертый общий элемент первого типа. Четвертый элемент ссылается на два элемента второго типа и т.д. «Элементарная ячейка» такой конструкции представлена на рис. 6.6.

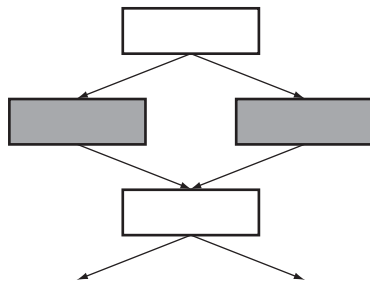


Рис. 6.6. Ячейка динамической структуры из четырех элементов разного типа

Задача 5. Написать программу для создания триарного дерева. В такой структуре каждый элемент ссылается на три элемента такого же типа. Каждый из этих элементов, в свою очередь, ссылается на три элемента и т.д. На рис. 6.7 представлена общая схема триарного дерева.

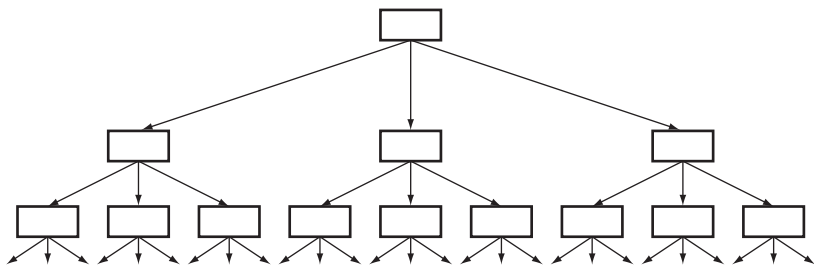


Рис. 6.7. Триарное дерево

Задача 6. Написать программу для создания N-кратного дерева. В такой структуре каждый элемент ссылается на N элементов такого же типа. Каждый из этих элементов, в свою очередь, ссылается на N элементов и т.д. Параметр N задается как константа. Поля-указатели структуры реализовать в виде массива.

Задача 7. Написать программу для создания динамической конструкции, состоящей из элементов двух типов. Элементы первого типа могут ссылаться на два элемента. Элементы второго типа могут ссылаться только на один элемент. В вершине иерархии находится элемент первого типа. Он ссылается на два элемента: один элемент первого типа и один элемент второго типа. Элемент первого типа ссылается на два элемента: первого и второго типа. Элемент второго типа ссылается на элемент второго типа. Общая схема создаваемой конструкции представлена на рис. 6.8.

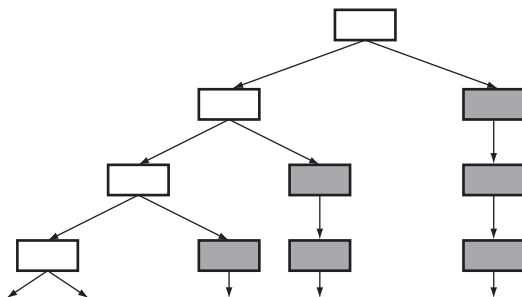


Рис. 6.8. Дерево из элементов двух типов

Задача 8. Написать программу для создания динамической конструкции из элементов двух типов. Элементы первого типа могут ссылаться на два элемента, а элементы второго типа могут ссылаться только на один элемент. Ссылки осуществляются через поля-указатели. Элементы организуются по следующему принципу. Начальный (первый) элемент относится к первому типу, и он ссылается на два элемента второго типа (второй и третий эле-

менты), каждый из которых ссылается на еще один элемент второго типа (четвертый и пятый). Четвертый и пятый элементы ссылаются на один общий элемент первого типа (шестой), который ссылается на два элемента второго типа, и т.д. «Элементарная ячейка» такой конструкции представлена на рис. 6.9.

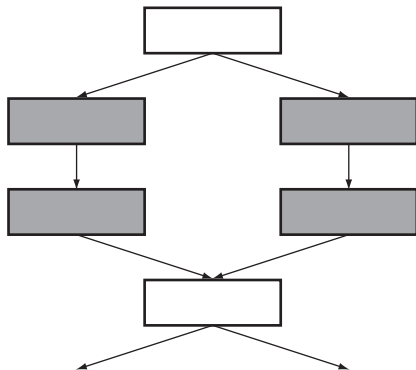


Рис. 6.9. Ячейка динамической структуры из шести разных типов

Задача 9. Написать программу для создания динамической конструкции из элементов двух типов («Список с ветвлением»). Элементы первого типа ссылаются на два элемента. Элементы второго типа не содержат полей для выполнения ссылок. Элементы первого типа образуют последовательный список. Каждый из элементов этого списка ссылается на следующий элемент и на элемент второго типа. Создаваемая конструкция проиллюстрирована на рис. 6.10.

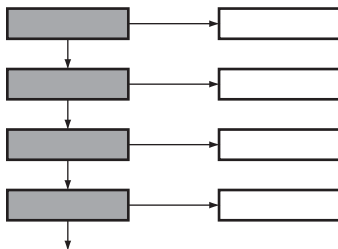


Рис. 6.10. «Список с ветвлением»

Задача 10. Написать программу для создания динамической конструкции из элементов двух типов («Двойной список с односторонней связью»). Элементы первого типа ссылаются на два элемента. Элементы второго типа ссылаются на один элемент. Элементы первого и второго типов образуют два последовательных списка. Каждый из элементов этих списков ссылается на следующий элемент. Кроме того, элементы первого списка (составленного из элементов первого типа) ссылаются на соответ-

ствующие элементы второго списка. Создаваемая конструкция проиллюстрирована на рис. 6.11.

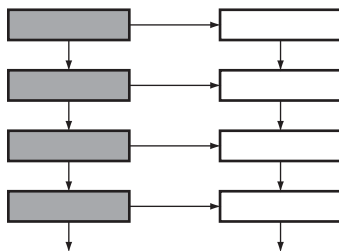


Рис. 6.11. «Двойной список с односторонней связью»

Задача 11. Написать программу для реализации комплексных чисел в алгебраической форме в виде элементов структуры. Предусмотреть функции для сложения, вычитания, деления и умножения комплексных чисел.

Задача 12. Написать программу для реализации комплексных чисел в тригонометрической форме в виде элементов структуры. Предусмотреть функции для сложения, вычитания, деления и умножения комплексных чисел.

Задача 13. Написать программу для реализации комплексных чисел в алгебраической и тригонометрической форме в виде элементов структуры. Предусмотреть функции для перевода числа из алгебраической формы в тригонометрическую, и наоборот.

Задача 14. Написать программу с функцией для вычисления косинуса от комплексного аргумента. Комплексные числа реализовать в виде элементов структуры. При вычислении воспользоваться формулой $\cos(z) = \cos(x)ch(y) - i \sin(x)sh(y)$, где $z = x + iy$.

Задача 15. Написать программу с функцией для вычисления синуса от комплексного аргумента. Комплексные числа реализовать в виде элементов структуры. При вычислении воспользоваться формулой $\sin(z) = \sin(x)ch(y) + i \cos(x)sh(y)$, где $z = x + iy$.

Задача 16. Написать программу с функцией для вычисления косинуса гиперболического от комплексного аргумента. Комплексные числа реализовать в виде элементов структуры. При вычислении воспользоваться формулой $ch(z) = ch(x)\cos(y) - ish(x)\sin(y)$, где $z = x + iy$.

Задача 17. Написать программу с функцией для вычисления синуса гиперболического от комплексного аргумента. Комплексные числа реализовать в виде элементов структуры. При вычислении воспользоваться формулой $sh(z) = sh(x)\cos(y) + ich(x)\sin(y)$, где $z = x + iy$.

Задача 18. Написать программу с функцией для вычисления тангенса от комплексного аргумента. Комплексные числа реализовать в виде элементов структуры. При вычислении воспользоваться формулой $tg(z) = \frac{\sin(z)}{\cos(z)}$, где $z = x + iy$ – комплексное число.

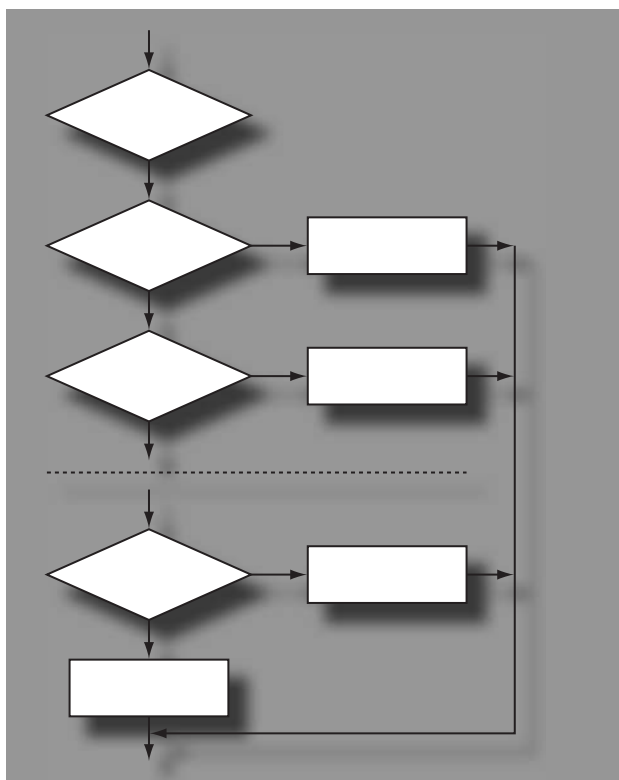
Задача 19. Написать программу с функцией для вычисления тангенса гиперболического от комплексного аргумента. Комплексные числа реализовать в виде элементов структуры. При вычислении воспользоваться формулой $th(z) = \frac{sh(z)}{ch(z)}$, где $z = x + iy$ – комплексное число.

Задача 20. Написать программу с функцией для вычисления логарифма от комплексного аргумента. Комплексные числа реализовать в виде элементов структуры. Логарифм вычислять следующим образом: $\ln(z) = \ln(r) + i\varphi$, где комплексное число в тригонометрической форме $z = r \cdot \exp(i\varphi)$, r – модуль комплексного числа, φ – аргумент.

Задача 21. Написать программу для вычисления суммы векторов в декартовом пространстве. Векторы реализовать в виде структуры.

Задача 22. Написать программу для решения уравнения вида $sh(x) = a$. Использовать структуры. Предусмотреть все возможные ситуации.

Объектно-ориентированное программирование в C++



ЧАСТЬ II

Глава 7

Классы и объекты

Объектно-ориентированное программирование является стандартом в технологии современного программирования. Основными понятиями данной парадигмы программирования являются класс и объект.

Классом называется описание некоторой структуры программы, обладающей набором внутренних переменных — **свойств**, и функций, имеющих доступ к свойствам — **методов**. Процесс объединения переменных и методов, в результате которого и получается класс, называется **инкапсуляцией**. Таким образом класс является базовой единицей инкапсуляции. Класс объединяет в себе как данные, так и методы для обработки этих данных.

Однако класс — это всего лишь описание, аналогичное описанию типа данных, и недоступное для прямого использования в программе. Для получения доступа к свойствам и методам класса необходимо создать **экземпляр класса**, называемый также **объектом**. К одному классу может принадлежать одновременно несколько объектов, каждый из которых имеет уникальное имя.

Объявление класса

Объявление класса начинается с ключевого слова `class`. Далее указывается имя класса. Поля и методы класса описываются в блоке в фигурных скобках. Причем как поля, так и методы могут быть закрытыми или открытыми. К закрытым членам класса можно обращаться только внутри класса, в то время как открытые члены доступны и за его пределами. Сначала перечисляются и определяются закрытые члены класса (поля и методы). Перед объявлением открытых членов указывается ключевое слово `public`. После закрывающих фигурных скобок объявления класса можно указать список объектов созданного класса. Таким образом, общий синтаксис объявления класса имеет вид:

```
class имя_класса{
    закрытые поля и методы класса
    public:
    открытые поля и методы класса
} список_объектов;
```

Пример простой программы с классом приведен в листинге 7.1.

Листинг 7.1. Программа с классом

```

#include "stdafx.h"
#include<iostream>
using namespace std;
// Объявление класса SimpleClass:
class SimpleClass{
    public:
    // Целочисленное поле класса:
    int number;};
int main(){
    // Создание объекта MyObj класса SimpleClass:
    SimpleClass MyObj;
    // Полю объекта присваивается значение:
    MyObj.number=5;
    cout<<"Object field value is "<<MyObj.number<<"\n";
    return 0;
}

```

В программе объявляется класс `SimpleClass`, в состав которого входит всего одно поле. Поле объявляется инструкцией `int number` точно так же, как переменная (в данном случае целочисленная). Поле является открытым, поэтому его объявлению предшествует ключевое слово `public` (после ключевого слова ставится двоеточие). После закрывающих блок класса фигурных скобок списка созданных объектов нет, поэтому сразу после закрывающей фигурной скобки ставится точка с запятой.

Непосредственно в методе `main()` инструкцией `SimpleClass MyObj` создается объект `MyObj` класса `SimpleClass`. Обращаем внимание, что имя класса, на основании которого создается объект, формально используется точно так же, как тип переменной при ее объявлении.

Хотя объект создан, его поле `number` является незаполненным. Чтобы полю `number` объекта `MyObj` присвоить значение, используем команду `MyObj.number=5`. Поле указывается после названия объекта и отделяется от него точкой. Это общий способ обращения к членам класса (таким же образом выполняется обращение к методам класса, но об этом речь пойдет несколько позже). После присваивания значения полю сообщение об этом выводится на экран. Обращение к полю `number` объекта `MyObj` выполняется в том же точечном формате, т.е. как `MyObj.number`.

Как отмечалось выше, объекты класса можно создавать сразу при объявлении класса, и этих объектов может быть несколько. Проиллюстрируем это, немного модифицировав уже рассмотренный программный код, как это показано в листинге 7.2.

Листинг 7.2. Создание нескольких объектов класса

```

#include "stdafx.h"
#include<iostream>
using namespace std;
// Объявление класса SimpleClass:
class SimpleClass{
public:
    // Целочисленное поле класса:
    int number;}
// Создание объектов MyObj1 и MyObj2 класса SimpleClass:
MyObj1, MyObj2;
int main(){
    // Полям объектов присваиваются значения:
    MyObj1.number=5;
    MyObj2.number=++MyObj1.number;
    cout<<"Object field value is "<<MyObj2.number<<"\n";
    return 0;
}

```

В данном случае создается два объекта `MyObj1` и `MyObj2` класса `SimpleClass`, причем соответствующие объекты создаются сразу после определения класса (поэтому в главном методе эти объекты создавать не нужно). Как и ранее, полю `number` объекта `MyObj1` присваивается значение 5, а поле `number` объекта `MyObj2` заполняется на основе значения поля `MyObj1.number`. В результате выполнения такой программы получим следующее:

```
Object field value is 6
```

Обращаем внимание на то, что оператор инкремента использован в префиксной форме. Это означает, что сначала происходит увеличение на единицу значения поля `MyObj1.number` и только после этого значение присваивается полю `MyObj2.number`.

Практически так же в классе объявляются методы. По большому счету метод – это функция, объявленная в классе. Общий синтаксис объявления метода в классе такой же, как и синтаксис объявления внешней функции:

```

тип_результата имя_метода (список_аргументов) {
    код метода
}

```

Однако между внешней (объявленной вне класса) функцией и методом класса существует принципиальное отличие. Заключается оно в том, что метод – это составная часть класса и для его вызова необходимо сначала создать соответствующий объект (исключение составляют статические методы, о которых речь будет идти позже). Кроме того, результат выполнения метода может (и, как правило, так и есть) зависеть от того, из какого объекта он вызывается. Обратимся к листингу 7.3.

Листинг 7.3. Класс с методами

```

#include<iostream>
using namespace std;
//Объявление класса SimpleClass:
class SimpleClass{
public:
    //Целочисленные поля класса:
    int m;
    int n;
    //Метод для вычисления суммы полей:
    int summa(){
        int k=n+m;
        return k;
    }
    //Метод для отображения значений полей:
    void show(){
        cout<<"m = "<<m<<endl;
        cout<<"n = "<<n<<endl;
    }
    //Метод для умножения полей на число:
    void mult(int k){
        n*=k;
        m*=k;
    }
};
int main(){
    //Создание объектов MyObj1 и MyObj2 класса SimpleClass:
    SimpleClass MyObj1,MyObj2;
    //Полям объектов присваиваются значения:
    MyObj1.m=1;
    MyObj1.n=2;
    MyObj2.m=8;
    MyObj2.n=9;
    //Сумма полей для разных объектов:
    cout<<"Total value for MyObj1 is "<<MyObj1.summa()<<endl;
    cout<<"Total value for MyObj2 is "<<MyObj2.summa()<<endl;
    //Умножение полей объектов на число:
    MyObj1.mult(3);
    MyObj2.mult(2);
    //Отображение значений полей объектов:
    MyObj1.show();
    MyObj2.show();
    return 0;
}

```

В данном случае в классе SimpleClass имеется два целочисленных (тип int) поля n и m, а также три метода: summa() для вычисления суммы полей

объекта, `show()` для отображения значений полей объекта и `mult()` для умножения полей объекта на число. Рассмотрим эти методы подробнее.

Метод `summa()` объявлен как такой, что имеет тип `int`, поэтому в качестве результата методом возвращается целочисленное значение. Как определяется возвращаемое методом значение, видно из его кода. В частности, вводится локальная целочисленная переменная `k`, значение которой определяется командой `k=n+m` и возвращается в качестве результата (команда `return k`). Стоит обратить внимание на два обстоятельства. Во-первых, методу «известны» поля объекта `n` и `m`, поэтому нет необходимости их описывать в методе еще раз (достаточно инструкции-описания полей). Во-вторых, аргументов у метода нет (но пустые круглые скобки все равно указываются, причем их надо будет указывать впоследствии и при вызове метода!). Кроме того, весь код можно было упростить, сведя к инструкции `return n+m` – переменная `k` введена для наглядности.

Метод `show()` не возвращает результат, поэтому в качестве его типа указано ключевое слово `void`. Как и в предыдущем случае, у метода нет аргументов, а его назначение состоит в отображении значений полей `n` и `m` объекта, из которого этот метод вызывается.

Наконец, `void`-метод `mult()` предназначен для умножения полей объекта, из которого вызывается метод, на число, которое указано аргументом метода. Изменение полей объекта реализуется с помощью команд `n*=k` и `m*=k`. Здесь `k` – целочисленный аргумент метода `mult()` (не следует путать этот аргумент с одноименной локальной переменной, объявленной в методе `summa()`!).

В главном методе программы командой `SimpleClass MyObj1, MyObj2` создаются два объекта класса `SimpleClass`, после чего полям объектов присваиваются значения (`MyObj1.m=1`, `MyObj1.n=2`, `MyObj2.m=8` и `MyObj2.n=9`). Далее вычисляется сумма полей каждого из объектов, для чего используются инструкции вида `MyObj1.summa()` и `MyObj2.summa()`, поля первого объекта умножаются на 3 (команда `MyObj1.mult(3)`), а поля второго объекта умножаются на 2 (команда `MyObj2.mult(2)`), и с помощью команд `MyObj1.show()` и `MyObj2.show()` значения полей выводятся на экран. Таким образом, в результате выполнения программы получим следующее:

```
Total value for MyObj1 is 3
Total value for MyObj2 is 17
m = 3
n = 6
m = 16
n = 18
```

Надо понимать, что при обращении методов к полям `n` и `m` используются поля именно того объекта, из которого вызывается соответствующий метод. Поэтому, например, в результате выполнения команды `MyObj1.show()` на экран выводятся значения полей объекта `MyObj1`, а командой `MyObj2.show()` отображаются поля объекта `MyObj2`, хотя формально в обоих случаях вызывается метод `show()` (но вызывается из разных объектов!).

Как и в случае внешних функций, при определении методов класса можно отделять прототип метода от непосредственного его описания. Главное отличие, по сравнению с внешними функциями, состоит в том, что впоследствии при описании метода класса вне пределов этого класса необходимо явно указывать, к какому классу принадлежит метод. Делается это посредством указания перед именем метода имени класса. Разделение имен осуществляется с помощью оператора разрешения области видимости `::`. Синтаксис описания метода класса вне класса таков (жирным шрифтом выделена часть кода, непосредственно относящаяся к объявлению и описанию метода класса):

```
class имя_класса{
    public:
        поля_класса
        методы_класса
        ...
        //Прототип определяемого метода:
        тип_результата имя_метода(список_аргументов);
        ...
};

...
int main(){
    ...
}

//Описание метода класса:
тип_результата имя_класса::имя_метода(список_аргументов) {
    код_метода
}
```

Например, корректным является программный код, приведенный в листинге 7.4.

Листинг 7.4. Описание метода класса вне класса

```
#include<iostream>
using namespace std;
class SimpleClass{
    public:
        int m;
        int n;
```

```

    //Объявление метода в классе (прототип метода):
    void show();
};
int main(){
    SimpleClass obj;
    obj.m=1;
    obj.n=2;
    obj.show();
    return 0;}
//Описание метода класса:
void SimpleClass::show(){
    cout<<"m = "<<m<<endl;
    cout<<"n = "<<n<<endl;}

```

Приведен упрощенный вариант предыдущего кода. В данном случае у класса `SimpleClass` два поля и всего один метод `show()`. Однако теперь в самом классе указан только прототип метода. Описание метода вынесено за пределы класса и имеет вид:

```

void SimpleClass::show(){
    cout<<"m = "<<m<<endl;
    cout<<"n = "<<n<<endl;}

```

Имя метода указано в формате `SimpleClass::show()` с явным указанием имени класса `SimpleClass`, которому принадлежит метод.

Приведенный способ описания методов класса, несмотря на кажущуюся избыточность, достаточно удобен и широко используется, особенно при наличии перекрестных вызовов и работе с большим числом классов.

Открытые и закрытые члены класса

Чтобы оценить чье-нибудь качество, надо иметь некоторую долю этого качества в самом себе.

В. Шекспир

Как отмечалось выше, члены класса могут быть закрытыми и открытыми. **По умолчанию члены класса считаются закрытыми.** Поэтому если для членов класса явно не указан спецификатор уровня доступа, они интерпретируются как закрытые. В рассмотренных примерах все члены класса были открытыми. Все они объявлены в блоке `public`, и доступ к ним осуществляется вне программного кода класса. Обычно же класс содержит как открытые, так и закрытые члены. Под членами, как и ранее, подразумеваются как поля, так и методы класса.

Закрытыми члены класса делают для ограничения несанкционированного внешнего доступа к этим членам, а также для упрощения процесса обработки объектов за счет формального уменьшения количества доступных атрибутов. Например, концепция класса подразумевает выполнение ограниченного количества операций с его полями. В этом случае разумно поля сделать закрытыми, а для доступа к ним создаются специальные методы (открытые члены класса), через которые и реализуется функциональность класса. Другой пример – когда в классе описываются методы, используемые только в программном коде класса при вызове других методов. В подобной ситуации также имеет смысл эти «вспомогательные» методы сделать закрытыми. Пример использования закрытых членов класса приведен в листинге 7.5.

Листинг 7.5. Использование закрытых членов класса

```
#include<iostream>
using namespace std;
class SimpleClass{
    //Закрытые члены класса:
    int m;
    int n;
public:
    //Открытые члены класса:
    void show();
    void setnm(int i,int j);
};

int main(){
    SimpleClass obj;
    obj.setnm(1,2);
    obj.show();
    return 0;}

//Описание методов класса:
void SimpleClass::show(){
    cout<<"m = "<<m<<endl;
    cout<<"n = "<<n<<endl;}
void SimpleClass::setnm(int i,int j){
    m=i;
    n=j;
}
```

Поля `n` и `m` класса `SimpleClass` объявлены как закрытые: они объявлены до блока `public`, по умолчанию такие члены класса, как отмечалось, являются закрытыми. В программном коде класса присутствует уже знакомый метод `show()`. Кроме этого метода, в классе объявлен метод `setnm()`, у которого два целочисленных аргумента. Метод необходим для того, чтобы задавать значения полей объектов класса. Дело в том, что поскольку поля `n` и `m` закры-

тые, присвоить им значения в главном методе программы тем способом, как это делалось ранее (т.е. через прямую ссылку на поле объекта) не получится. Наличие в главном методе `main()` инструкции вида `obj.n` или `obj.m` (`obj` – имя объекта) приведет к ошибке. В программе заполнение полей осуществляется командой `obj.setnm(1, 2)`. Поскольку метод `setnm()` является открытым, данная команда корректна. В рамках этого метода осуществляется обращение к полям `n` и `m`. Но теперь ошибки нет, поскольку метод класса, каков бы он ни был (открытый или закрытый), может обращаться не только к открытым, но и к закрытым полям. Аналогично обстоят дела и с методом `show()`. Этот метод также обращается к закрытым полям и отображает их значение. Результат выполнения программы следующий:

```
m = 1
n = 2
```

При описании закрытых членов класса можно использовать ключевое слово `private`. Например, рассмотренный выше класс может быть объявлен следующим образом:

```
class SimpleClass{
    private:
        int m,n;
    public:
        void show();
        void setnm(int i,int j);
};
```

Здесь также одной командой `int m,n` объявлены оба поля `n` и `m` – для полей одного типа такой синтаксис допустим.

Статические члены класса

Хотя каждый объект класса имеет одинаковый набор полей и методов, значения полей у каждого объекта свои, а результат вызова методов в общем случае зависит от того, из какого объекта метод вызывается. На самом деле, это не всегда так. Существуют особые члены класса, которые называют статическими. **Статический член является общим для всех объектов этого класса.**

Необходимость в таких статических членах продиктована рядом практических соображений, и их использование во многих случаях оправдано. Например, в программе нужно контролировать количество созданных на данный момент объектов определенного класса. В этом случае создается статический член класса, значение которого определяется количеством объектов в работе. В отличие от обычного поля класса, статическое поле не исчезает при удалении объекта. В некотором смысле статические перемен-

ные напоминают глобальные переменные программы. Однако статические переменные, в отличие от глобальных переменных, полностью согласуются с принципами объектно-ориентированного программирования, и в частности с принципом инкапсуляции.

Статические члены объявляются, как и обычные, но перед статическим членом указывается ключевое слово `static`. Рассмотрим сначала случай статического поля. Пример приведен в листинге 7.6.

Листинг 7.6. Статические поля

```
#include<iostream>
using namespace std;
class SimpleClass{
public:
    //Статическое поле:
    static int m;
    //Нестатическое поле:
    int n;
    void show();
}obj1,obj2;
//Повторное объявление переменной:
int SimpleClass::m;
int main(){
    SimpleClass::m=10;
    obj1.n=1;
    obj2.n=2;
    obj1.show();
    obj2.show();
    obj1.m=100;
    obj2.show();
    return 0;}
//Описание метода:
void SimpleClass::show(){
    cout<<"Static field m = "<<m<<endl;
    cout<<"Nonstatic field n = "<<n<<endl;}
```

Как и прежде, класс `SimpleClass` содержит два целочисленных поля `m` и `n` и метод `show()` для отображения значения полей. Теперь одно из полей класса объявлено как статическое (инструкция `static int m`). Кроме того, есть еще одна особенность. Она связана с тем, что объявление статической переменной в классе не означает выделения памяти под эту переменную. Чтобы под статическое поле класса было выделено место, вне класса выполняют объявление `int SimpleClass::m`. От объявления глобальной переменной отличие состоит в том, что для поля `m` явно указывается принадлежность к классу `SimpleClass`. С обычными, нестатическими, полями таких проблем не возникает, поскольку обычные поля существуют

только при наличии объекта, и у каждого объекта поля свои (хотя и называются одинаково). При создании объекта в памяти выделяется место под его поля. Статические поля класса в некотором смысле существуют независимо от объектов класса и создаются до создания объектов класса. Поэтому для них в явном виде выделяется место в памяти.

В главном методе программы командой `SimpleClass::m=10` статическому полю присваивается значение. Для обращения к полю используется имя класса, а не объект, как обычно. Более того, при объявлении статического поля ему присваивается нулевое значение. Поэтому формально полю `m` значение можно было и не присваивать.

Командами `obj1.n=1` и `obj2.n=2` нестатическим полям объектов присваиваются значения. Значения нестатических полей для каждого объекта уникальны. А статическое поле на все объекты одно. В последнем несложно убедиться, воспользовавшись командами `obj1.show()` и `obj2.show()`.

К статическому полю можно обращаться и через объект. Например, корректной является команда `obj1.m=100`, которой статическому полю `m` присваивается значение 100. Хотя формально меняется поле объекта `obj1`, изменения затронут все объекты, поскольку поле статическое. Проверка этого замечательного факта осуществляется командой `obj2.show()`. Результат выполнения команды таков:

```
Static field m = 10;
Nonstatic field n = 1
Static field m = 10;
Nonstatic field n = 2
Static field m = 100;
Nonstatic field n = 2
```

Подобно статическим полям объявляются и статические методы. Однако по сравнению с обычными методами класса, на статические, в силу объективных причин, накладываются существенные ограничения. Так, статические методы напрямую (т.е. не через аргументы) могут ссылаться на статические члены класса. Статические методы при перегрузке не могут иметь нестатические версии, они не переопределяются при наследовании (не могут быть виртуальными), не получают указатель `this` на объект класса, и имеют ряд других особенностей. Пример использования статического метода приведен в листинге 7.7.

Листинг 7.7. Статические методы

```
#include<iostream>
using namespace std;
class SimpleClass{
```

```

public:
//Статическое поле:
static int m;
//Нестатическое поле:
int n;
void show();
//Статический метод:
static void msum(int k);
}obj1,obj2;
//Повторное объявление переменной:
int SimpleClass::m;
int main(){
//Вызов статического метода:
SimpleClass::msum(10);
obj1.n=1;
obj2.n=2;
obj1.show();
obj2.show();
//Вызов статического метода:
obj1.msum(90);
obj2.show();
return 0;}
//Описание нестатического метода:
void SimpleClass::show(){
    cout<<"Static field m = "<<m<<endl;
    cout<<"Nonstatic field n = "<<n<<endl;}
//Описание статического метода:
void SimpleClass::msum(int k){
    m+=k;}

```

Приведенный пример является видоизмененным кодом из листинга 7.6. В нем появился статический метод `msum()`, которым к статическому полю `m` класса прибавляется число, указанное аргументом функции. Обращаем внимание, что этим методом изменить значение нестатического поля `n` не удастся, поскольку поле `n` не является статическим. Ведь статический метод общий для всех объектов класса и может вызываться без конкретного объекта, а для изменения нестатического поля необходим объект.

В главном методе первой выполняется команда `SimpleClass::msum(10)`, которой статическое поле получает значение 10 (начальное значение по умолчанию 0, и оно увеличивается на 10). В этой команде статический метод вызывается через ссылку на класс `SimpleClass`. Как и в случае со статическим полем, статический метод можно вызывать и через объект класса, как, например, в команде `obj1.msum(90)`. Результат выполнения команды такой же, как и в предыдущем случае.

Перегрузка методов

Это и серьезно, и в то же время как-то мобилизует.

Из к/ф «Карнавальная ночь»

Как и обычные функции, методы классов можно перегружать. В этом случае создается несколько вариантов одного и того же метода, но с разными прототипами. Отличие может быть связано с разными типами возвращаемых результатов или с разным типом и количеством аргументов. Формально различные варианты перегруженного метода могут быть, с точки зрения их функциональности, абсолютно разными. Но хорошим стилем считается, если разные варианты перегруженного метода объединены общей идеей. Такой подход позволяет использовать единый интерфейс и при этом учесть особенности вызова соответствующего метода с разным набором аргументов. Напомним, что данная концепция получила название полиморфизма, одного из трех фундаментальных механизмов, на которых базируется объектно-ориентированное программирование.

Пример перегрузки методов класса приведен в листинге 7.8.

Листинг 7.8. Перегрузка методов

```
#include <iostream>
using namespace std;
class MyClass{
    int a,b;
public:
    //Перегруженный метод:
    void setab(int i,int j){
        a=i;
        b=j;
    }
    void setab(int i){
        a=i;
        b=i;
    }
    void getab(){
        cout<<"a = "<<a<<endl;
        cout<<"b = "<<b<<endl;
    }
}obj1,obj2;
int main(){
    obj1.setab(1,2);
    obj2.setab(3);
    obj1.getab();
    obj2.getab();
    return 0;
}
```

В программе создан класс `MyClass`, в котором два закрытых целочисленных поля `a` и `b`, метод `getab()` для отображения значения закрытых полей и перегруженный метод `setab()` для записи значений в закрытые поля. У метода `setab()` два варианта: с одним целочисленным аргументом и с двумя аргументами. Если методу передано два аргумента, то эти аргументы присваиваются в качестве значений полям `a` и `b`. Если у метода аргумент один, то соответствующее значение присваивается каждому полю `a` и `b`. Благодаря этому в главном методе программы можно использовать как команду `obj1.setab(1, 2)`, так и команду `obj2.setab(3)`. В зависимости от количества переданных методу `setab()` аргументов вызываются разные версии этого метода. Результат выполнения программы имеет вид:

```
a = 1
b = 2
a = 3
b = 3
```

О перегрузке методов речь будет идти много раз, особенно в контексте механизма переопределения методов при наследовании и при перезагрузке конструкторов.

В заключение заметим, что классы можно объявлять внутри других классов. В этом случае говорят о вложенных классах. Правда, такой подход на практике используется крайне редко. Также объявление класса может содержаться внутри функции. Такой класс является внутренним и доступен только в соответствующей функции.

Примеры решения задач

Трудные задачи выполняем медленно, невозможные – чуть погодя.

Девиз ВВС США

Здесь рассматриваются некоторые задачи, при решении которых применяются принципы объектно-ориентированного программирования.

■ Вычисление логарифма

В листинге 7.9 приведен пример программного кода, в котором для вычисления натурального логарифма по формуле

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots + \frac{(-1)^{n+1}x^n}{n} + \dots = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}x^n}{n}$$

создается специальный класс. У класса два поля и метод для вычисления логарифма. Что касается полей, то в одно поле (поле `double x`) заносится значение аргумента x для выражения $\ln(1+x)$, а второе поле (поле `int N`) определяет верхнюю границу N суммирования в выражении $\ln(1+x) \approx \sum_{n=1}^N \frac{(-1)^{n+1} x^n}{n}$ (таким образом, у метода класса `Ln()` аргумента нет).

Листинг 7.9. Вычисление логарифма

```
#include <iostream>
#include <cmath>
using namespace std;
class MyLog{
public:
double x;
int N;
double Ln(){
double s=0,q=x;
int i;
for(i=1;i<=N;i++){
s+=q/i;
q*=-x;}
return s;}
};
int main(){
MyLog a;
cout<<"Enter x = ";
cin>>a.x;
cout<<"Enter N = ";
cin>>a.N;
cout<<"ln("<<1+a.x<<" ) = "<<a.Ln()<<endl;
cout<<"ln("<<1+a.x<<" ) = "<<log(1+a.x)<<endl;
return 0;}
```

Результат выполнения программы может иметь следующий вид (жирным шрифтом выделен ввод пользователя):

```
Enter x = 0.5
Enter N = 10
ln(1.5) = 0.405435
ln(1.5) = 0.405465
```

Для сравнения в последней строке приведен результат вычисления натурального логарифма с помощью встроенной функции.

■ Модуль и аргумент комплексного числа

Создадим класс для реализации комплексных чисел. Класс содержит два поля – действительная и мнимая части комплексного числа, а также методы для вычисления модуля и аргумента комплексного числа. Последние две характеристики вычисляются на основании значения полей класса. Напомним, что в алгебраической форме записи любое комплексное число может быть представлено в виде $z = x + iy$, где действительные числа $x = \operatorname{Re}(z)$ и $y = \operatorname{Im}(z)$ соответственно действительная и мнимая части комплексного числа z , мнимая единица $i^2 = -1$. Это же число может быть записано в тригонометрическом представлении как $z = r \exp(i\varphi)$, где модуль комплексного числа $r = \sqrt{x^2 + y^2}$, а аргумент φ таков, что $\sin(\varphi) = y/r$ и $\cos(\varphi) = x/r$. Программный код, в котором описан класс для реализации комплексных чисел, представлен в листинге 7.10.

Листинг 7.10. Модуль и аргумент комплексного числа

```
#include <iostream>
#include <cmath>
using namespace std;
class Compl{
public:
double Re;
double Im;
double modul() {
double r;
r=sqrt(Re*Re+Im*Im);
return r;}
double argument() {
double phi;
phi=atan2(Re,Im);
return phi;}
};
int main() {
Compl z;
z.Re=sqrt(3);
z.Im=-1;
cout<<"|z| = "<<z.modul()<<endl;
cout<<"phi = "<<z.argument()<<endl;
return 0;}
```

О комплексных числах речь еще будет идти, в частности, при рассмотрении вопроса о переопределении операторов.

■ Схема Бернулли

Схемой Бернулли называется последовательность независимых испытаний с вероятностью «успеха» в одном испытании, равном p . Если серия состоит из n опытов, то вероятность того, что будет иметь место ровно m «успехов», дается выражением $P_n(m) = C_n^m p^m q^{n-m}$, где биномиальный коэффициент $C_n^m = \frac{n!}{m!(n-m)!}$, а вероятность «неудачи» $q = 1 - p$.

Создадим класс, в котором вероятность успеха в одном опыте будет определяться значением поля класса, результат отдельного «опыта» будет моделироваться специальным методом. В качестве значения методом возвращается значение 0 («неудача» с вероятностью $q = 1 - p$) и 1 («успех» с вероятностью p). Для определения результата (количества успехов) в серии опытов упомянутый метод переопределим так, что если ему передается аргумент (целое число, определяющее количество опытов в серии), то методом возвращается количество успехов в такой серии. Соответствующий программный код представлен в листинге 7.11.

Листинг 7.11. Схема Бернулли

```
#include <iostream>
#include <cstdlib>
using namespace std;
class Bernoulli{
public:
double p;
int test(){
    int N=1000;
    int xi;
    xi=rand()%N;
    if(xi<p*N) return 1;
    else return 0;}
int test(int n){
    int s=0,i;
    for(i=1;i<=n;i++) s+=test();
    return s;}
};
int main(){
    int i;
    Bernoulli obj;
    obj.p=0.3;
    for(i=1;i<=9;i++) cout<<obj.test()<<" ";
```



```
cout<<endl;
for(i=1;i<=9;i++) cout<<i*100<<": "<<obj.test(i*100)<<endl;
return 0;}
```

Программой сначала проводится серия из 9 опытов – при успехе отображается 1, а при неудаче отображается 0. Далее отображается количество успехов в серии из 100, 200 и т.д. до 900 включительно опытов. Результат может иметь следующий вид:

```
1 0 0 0 1 0 0 0 0
100: 21
200: 55
300: 100
400: 125
500: 125
600: 205
700: 203
800: 252
900: 286
```

Стоит обратить внимание, что при перегрузке метода `test()` (версия метода с целочисленным аргументом) используется вызов этого же метода, но без аргумента. О рекурсии в данном случае речь не идет, поскольку вызывается иная версия метода. Также обращаем внимание читателя, что в данном случае вычисляется «экспериментальное» количество успехов, поэтому от запуска к запуску результат выполнения программы может меняться. Читатель может взять себе на заметку, что в пределе при увеличении количества опытов в серии до бесконечности отношение количества успехов к количеству опытов в серии стремится к величине p (математическое ожидание количества успехов в схеме Бернулли из n опытов равняется np). В данном случае вероятность успеха $p = 0.3$. Правда, чтобы получить частоту успехов, близкую к указанному значению, необходимо выбрать число для количества опытов в схеме на несколько порядков больше, чем те, что использовались в программе.

■ Метод последовательных итераций

Вернемся к задаче по вычислению корня уравнения $x = \varphi(x)$ методом последовательных итераций. Для этого создадим класс, полями которого будут начальное приближение и общее число итераций. У класса также будет два метода: метод, определяющий зависимость $\varphi(x)$, и метод, с помощью которого реализуется итерационный процесс $x_{n+1} = \varphi(x_n)$. Программный код приведен в листинге 7.12.

Листинг 7.12. Метод последовательных итераций

```

#include <iostream>
#include <cmath>
using namespace std;
class Eqns{
public:
double x0;
int n;
double phi(double x){
return 0.8*cos(x);}
double root(){
double s=x0;
int i;
for(i=1;i<=n;i++) s=phi(s);
return s;}
};
int main(){
Eqns obj;
obj.x0=0;
obj.n=100;
cout<<"x = "<<obj.root()<<endl;
return 0;}

```

В данном случае решается уравнение $x = 0.8 \cos(x)$. Программой вычисляется следующее значение для корня:

$$x = 0.641134$$

Результат получен для 100 итераций с нулевым начальным приближением для корня уравнения.

■ Полет тела

Решим задачу о полете тела, брошенного под углом к горизонту, с помощью методов объектно-ориентированного программирования. Здесь создадим специальный класс, полями которого будут начальная скорость и угол, под которым тело брошено к горизонту. Методами класса будут вычисляться координаты (горизонтальная и вертикальная) проекции скорости на координатные оси. Программный код приведен в листинге 7.13.

Листинг 7.13. Полет тела

```

#include <iostream>
#include <cmath>
using namespace std;
const double g=9.8;

```

```

const double pi=3.1415;
class Body{
    double T(){
        return 2*V0*sin(alpha)/g;}
public:
    double V0;
    double alpha;
    double x(double t){
        if(t<=T()) return V0*cos(alpha)*t;
        else return V0*cos(alpha)*T();}
    double y(double t){
        if(t<=T()) return V0*sin(alpha)*t-g*t*t/2;
        else return 0;}
    double Vx(double t){
        if(t<=T()) return V0*cos(alpha);
        else return 0;}
    double Vy(double t){
        if(t<=T()) return V0*sin(alpha)-g*t;
        else return 0;}
    void show(double t){
        printf("%s%f%s%9f%s%9f%s%9f%s%9f%s", "t=", t, ": x=", x(t),
            " y=", y(t), " Vx=", Vx(t), " Vy=", Vy(t), "\n");}
};

int main(){
    int i;
    Body obj;
    obj.V0=10;
    obj.alpha=pi/3;
    for(i=0;i<10;i++) obj.show(0.2*i);
    return 0;}

```

В программе объявлены две глобальные константы – число π и ускорение свободного падения. Основной функциональный блок программы реализуется через класс `Body`. В этом классе описан закрытый метод `T()`, которым на основе полей `V0` (начальная скорость тела) и `alpha` (угол, под которым тело брошено к горизонту) типа `double` вычисляется время полета тела. Этот важный показатель необходим для вычисления положения тела в произвольный момент времени. Координаты тела и скорость (проекции на координатные оси) тела вычисляются методами `x()`, `y()`, `Vx()` и `Vy()`. При вычислениях использованы следующие соотношения: время полета тела, брошенного с начальной скоростью V_0 под углом α к горизонту $T = \frac{2V_0 \sin(\alpha)}{g}$, проекция скорости тела на горизонтальную координатную ось $V_x(t) = V_0 \cos(\alpha)$ при $t < T$ и $V_x(t) = 0$ при $t > T$ (тело упа-

ло), на вертикальную ось $V_y(t) = V_0 \sin(\alpha) - gt$ при $t < T$ и $V_y(t) = 0$ при $t > T$ (тело упало), координаты тела $x(t) = V_0 \cos(\alpha)t$ при $t < T$ и $x(t) = V_0 \cos(\alpha)T$ при $t > T$, $y(t) = V_0 \sin(\alpha)t - \frac{gt^2}{2}$ при $t < T$ и

$y(t) = 0$ при $t > T$. В главном методе программы выводится таблица значений координат и скоростей для разных моментов времени. Результат выполнения программы имеет следующий вид:

```
t=0.000000: x= 0.000000 y= 0.000000 vx= 5.000267 vy= 8.660100
t=0.200000: x= 1.000053 y= 1.536020 vx= 5.000267 vy= 6.700100
t=0.400000: x= 2.000107 y= 2.680040 vx= 5.000267 vy= 4.740100
t=0.600000: x= 3.000160 y= 3.432060 vx= 5.000267 vy= 2.781100
t=0.800000: x= 4.000214 y= 3.792080 vx= 5.000267 vy= 0.820100
t=1.000000: x= 5.000267 y= 3.760100 vx= 5.000267 vy=-1.139900
t=1.200000: x= 6.000321 y= 3.336120 vx= 5.000267 vy=-3.099900
t=1.400000: x= 7.000374 y= 2.520139 vx= 5.000267 vy=-5.059900
t=1.600000: x= 8.000428 y= 1.312159 vx= 5.000267 vy=-7.019900
t=1.800000: x= 8.837309 y= 0.000000 vx= 0.000000 vy= 0.000000
```

Стоит обратить внимание на использование в данном случае для вывода данных функции `printf()`. Первым аргументом функции указана строка, определяющая формат вывода. В этой строке инструкция `%s` означает вывод текстового блока, инструкция `%f` используется для индикации вывода числа с плавающей точкой. Цифра между символом процента `%` и буквой `f` определяет минимальное количество позиций для вывода числа. Сами выводимые на экран блоки (текст и числа) перечислены через запятую следующими после первого аргументами функции `printf()`.

Резюме

Я всегда завидовал, Штирлиц, вашему умению выстраивать четкую логическую направленность.

Из к/ф «Семнадцать мгновений весны»

1. В основе принципа инкапсуляции, одного из основополагающих принципов объектно-ориентированного программирования, лежит концепция классов и объектов. Класс является базовой единицей инкапсуляции и объединяет в себе как данные, так и программный код для обработки этих данных. Конкретный экземпляр класса называется объектом.

2. Объявление класса начинается с ключевого слова `class`, содержит уникальное имя класса и, в фигурных скобках, описание полей и методов класса. Поля и методы класса называются членами класса.
3. Члены класса могут быть открытыми и закрытыми. По умолчанию члены класса являются закрытыми. При описании открытых членов используют идентификатор `public`.
4. К открытым членам класса существует внешний доступ (к ним можно обращаться вне класса). Закрытые члены класса доступны только в пределах класса.
5. Для обращения в программе к полю или методу объекта используют «точечный» синтаксис: указывается имя объекта и, через точку, имя поля или метода. Имя метода завершается парой круглых скобок, даже если методу не передаются аргументы.
6. Статические члены класса отличаются от обычных тем, что статическое поле или метод являются общими для всех объектов класса. При объявлении статических членов используют ключевое слово `static`.
7. Как и обычные функции, методы класса могут перегружаться. В этом случае описывается несколько вариантов метода с разными прототипами, но одним и тем же названием.

Контрольные вопросы

Ложь – удел рабов. Свободные люди должны говорить правду.

III. Монтень

1. Что такое класс и объект? Чем класс отличается от объекта?
2. Как описывается класс и как создаются объекты?
3. Что такое поля и методы класса? Как они объявляются?
4. Чем закрытые члены класса отличаются от открытых?
5. Что такое статическое поле? В чем его особенности?
6. Что такое перегрузка методов и как она выполняется?

Задачи для самостоятельного решения

Задача 1. Написать программу для вычисления косинуса

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}.$$

Необходимо создать специальный класс с соответствующим методом. Аргумент x и верхняя граница ряда определяются как поля класса. Рассмотреть случай, когда поле, определяющее границу ряда, является статическим.

Задача 2. Написать программу для вычисления синуса

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}.$$

Необходимо создать специальный класс с соответствующим методом. Аргумент x и верхняя граница ряда определяются как поля класса. Рассмотреть случай, когда поле, определяющее границу ряда, является статическим.

Задача 3. Написать программу для вычисления экспоненты

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

Необходимо создать специальный класс с соответствующим методом. Аргумент x и верхняя граница ряда определяются как поля класса. Рассмотреть случай, когда поле, определяющее границу ряда, является статическим.

Задача 4. Написать программу для вычисления гиперболического коси-

нуса $ch(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}$. Необходимо создать специальный класс с соответствующим методом. Аргумент x и верхняя граница ряда определяются как поля класса. Рассмотреть случай, когда поле, определяющее границу ряда, является статическим.

Задача 5. Написать программу для вычисления гиперболического синуса

$sh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}$. Необходимо создать специальный класс с соответствующим методом. Аргумент x и верхняя граница ряда определяются как поля класса. Рассмотреть случай, когда поле, определяющее границу ряда, является статическим.

Задача 6. Написать программу для вычисления ряда

$$\frac{\sin(x)}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n+1)!}.$$

Необходимо создать специальный класс с соответствующим методом. Аргумент x и верхняя граница ряда определяются как поля класса. Рассмотреть случай, когда поле, определяющее границу ряда, является статическим.

Задача 7. Написать программу для вычисления факториала числа $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$. Использовать специальный класс, полем которого является число n , а факториал вычисляется методом класса.

Задача 8. Написать программу для вычисления двойного факториала числа $n!! = n \cdot (n-2) \cdot (n-4) \cdot \dots$. Использовать специальный класс, полем которого является число n , а двойной факториал вычисляется методом класса.

Задача 9. Написать программу для вычисления модуля вектора в трехмерном декартовом пространстве. Вектор реализовать в виде объекта класса, поля которого являются элементами (координатами) вектора. Для вычисления модуля вектора создать специальный метод.

Задача 10. Написать программу для вычисления корней квадратного уравнения вида $ax^2 + bx + c = 0$. Коэффициенты a , b и c реализовать в виде полей класса. Корни уравнения вычислять с помощью одного переопределенного метода или с помощью двух разных методов (отдельно для меньшего и большего корня). Предусмотреть все возможные варианты, в том числе и когда корни комплексные.

Задача 11. Написать программу для вычисления точек экстремума полинома третьей степени. Коэффициенты, задающие полином, реализовать в виде полей класса. Для вычисления экстремумов использовать метод (или методы) класса.

Задача 12. Написать, путем создания специального класса, программу для решения уравнения вида $a \sin(x) + b \cos(x) = c$.

Задача 13. Написать программу для решения статистическими методами следующей задачи. Из зенитного орудия производится стрельба по самолету. Вероятность попадания из одного выстрела равна p . Стрельба производится до первого попадания. В программе необходимо создать класс с полем p и методом для вычисления количества выстрелов до первого попадания включительно. Предусмотреть метод для вычисления среднего значения количества выстрелов до поражения цели (в пределе это значение стремится к величине $1/p$).

Задача 14. Написать программу для решения статистическими методами следующей задачи. Из зенитного орудия производится стрельба по самолету. Вероятность попадания из одного выстрела равна p . Стрельба производится n раз или до первого попадания. В программе необходимо создать класс с полем p и методом (n – аргумент метода) для вычисления результата стрельбы (0 если самолет не сбит, и 1 если цель уничтожена). Предусмотреть метод для вычисления оценочного значения вероятности поражения цели (в пределе это значение стремится к величине $1 - (1 - p)^n$).

Задача 15. Написать программу для решения уравнения методом половинного деления. Для этого создать класс. Функцию, определяющую уравнение, и процедуру поиска корня реализовать в виде методов класса. Границы интервала поиска корня реализовать в виде полей класса.

Задача 16. Написать программу для решения уравнения методом хорд. Для этого создать класс. Функцию, определяющую уравнение, и процедуру поиска корня реализовать в виде методов класса. Границы интервала поиска корня реализовать в виде полей класса.

Задача 17. Написать программу для решения уравнения методом касательных (метод Ньютона). Для этого создать класс. Функцию, определяющую уравнение, и процедуру поиска корня реализовать в виде методов класса. Начальное приближение для корня реализовать в виде поля класса.

Задача 18. Создать класс с перегруженным методом для решения уравнения методом половинного деления либо методом хорд (например, в зависимости от наличия или типа переданных аргументов).

Задача 19. Создать класс с перегруженным методом для решения уравнения методом последовательных итераций либо методом Ньютона (например, в зависимости от наличия или типа переданных аргументов).

Задача 20. Написать программу с классом, в которой по максимальной высоте подъема H (поле класса) и дальности полета L (поле класса) определяется начальная скорость тела V (метод класса) и угол α (метод класса), под которым тело брошено к горизонту. Воспользоваться соотношениями $tg(\alpha) = 4H/L$ и $V = \sqrt{gL/\sin(2\alpha)}$.

Глава 8

Работа с объектами

*Во всякую эпоху действует одна
какая-нибудь идея, сообщающая
известную форму событиям
и определяющая их конечный ре-
зультат.*

Г. Бокль

В этой главе более детально остановимся на тех подходах и приемах, которые широко используются при работе с объектами в рамках концепции объектно-ориентированного программирования.

Передача объектов аргументами

Объекты могут передаваться аргументами функциям и методам, как и обычные переменные (т.е. переменные базовых типов). В качестве типа переменной-объекта в этом случае указывается имя класса, к которому принадлежит соответствующий объект. Пример передачи в качестве аргументов объектов функциям приведен в листинге 8.1.

Листинг 8.1. Передача объектов аргументом функции

```
#include <iostream>
using namespace std;
//Первый класс:
class ClassA{
public:
double x;
double y;
}objA;
//Второй класс:
class ClassB{
public:
int a;
int b;
//Методу объектом передается объект:
double f(ClassA obj){
return a*obj.x+b*obj.y;
}
}objB;
```

```
//Внешняя функция с аргументом-объектом:
void showB(ClassB obj){
    cout<<"a = "<<obj.a<<endl;
    cout<<"b = "<<obj.b<<endl;
}
int main(){
    objA.x=2.5;
    objA.y=3.6;
    objB.a=2;
    objB.b=5;
    cout<<"Value is "<<objB.f(objA)<<endl;
    showB(objB);
    return 0;
}
```

В программе объявляется два класса и одна внешняя функция. Первый класс `ClassA` имеет два поля `x` и `y` типа `double`. У второго класса `ClassB` два целочисленных поля `a` и `b` типа `int` и метод `f()`, который в качестве значения возвращает число типа `double`. Этот метод интересен тем, что его аргументом является объект класса `ClassA`. Аргумент объявлен инструкцией `ClassA obj`. В качестве значения методом возвращается выражение `a*obj.x+b*obj.y`, т.е. сумма произведений соответствующих полей объекта, из которого вызывается метод, и объекта, переданного методу в качестве аргумента.

Внешняя функция `showB()` результат не возвращает, а аргументом ей передается объект класса `ClassA`. Для этого объекта методом отображаются значения полей `a` и `b`. Результат выполнения программы имеет вид:

```
Value is 23
a = 2
b = 5
```

Передача аргументами объектов – подход достаточно эффективный, поскольку позволяет обрабатывать целиком объекты, а не передавать каждое поле отдельно.

Возвращение результатом объектов

Так же легко, как передача аргументами объектов, **объекты возвращаются в качестве результата функции. В качестве типа результата функции указывается имя класса, объект которого возвращается в качестве результата функции или метода.** Следует также помнить, что поскольку возвращается объект, в функции или методе необходимо создать временный, локальный объект такого же типа, как результат. Пример приведен в листинге 8.2.

Листинг 8.2. Возвращение в качестве результата объекта

```

#include <iostream>
using namespace std;
class ClassA{
public:
double x;
double y;
}objA;
class ClassB{
public:
int a;
int b;
}objB;
ClassA sumAB(ClassA obj1,ClassB obj2){
ClassA tmp;
tmp.x=obj1.x+obj2.a;
tmp.y=obj1.y+obj2.b;
return tmp;
}
int main(){
ClassA obj;
objA.x=2.5;
objA.y=3.6;
objB.a=2;
objB.b=5;
//Результатом функции является объект:
obj=sumAB(objA,objB);
cout<<"x = "<<obj.x<<endl;
cout<<"y = "<<obj.y<<endl;
return 0;
}

```

В результате выполнения программы получим

```

x = 4.5
y = 8.6

```

Как и ранее, программа содержит два класса: в классе `ClassA` два поля `x` и `y` типа `double`, в классе `ClassB` два поля `a` и `b` типа `int`. В программе объявлена единственная функция `sumAB()`: у нее два аргумента – объекты классов `ClassA` и `ClassB`, а в качестве результата функцией возвращается объект класса `ClassA`.

В теле функции создается локальный объект `tmp` класса `ClassA`. Командами `tmp.x=obj1.x+obj2.a` и `tmp.y=obj1.y+obj2.b` определяются значения полей этого объекта, после чего объект `tmp` возвращается в качестве значения функции (инструкция `return tmp`).

В главном методе программы объекту `obj` класса `ClassA` в качестве значения присваивается результат вызова функции `sumAB()` (команда `obj=sumAB(objA, objB)`). Значения полей определенного таким образом объекта выводятся на экран.

Указатели на объекты

Тут без колдовства не обошлось...

Из к/ф «Чародеи»

Как и для обычных переменных базовых типов, **для объектов могут создаваться указатели. Значением переменной-указателя является адрес первой ячейки области памяти, выделенной под объект.** Создаются указатели на объекты так же, как и указатели на переменные базовых типов данных: указывается имя класса, на объект которого создается указатель, имя переменной-указателя, которому предшествует оператор `*`.

Чтобы получить адрес памяти, по которому записан объект, перед именем объекта указывают оператор `&`. Если перед именем переменной-указателя на объект указать оператор `*`, получим значение переменной, на которую ссылается указатель, т.е. получаем доступ непосредственно к объекту. Кроме того, к полям и методам объекта можно получать доступ непосредственно через указатель. Для этого используют оператор «стрелка» `->`: после указателя через стрелку указывается имя поля или метода. Пример создания и использования указателей на объекты приведен в листинге 8.3.

Листинг 8.3. Указатели на объекты

```
#include <iostream>
using namespace std;
class MyClass{
public:
    int n;
    void show(){
        cout<<"n = "<<n<<endl;}
};
int main(){
    MyClass a,b;
    //Указатели на объект:
    MyClass *p,*q;
    //Значения указателей:
    p=&a;
    q=&b;
```

```
//Доступ к полям и методам через указатель:
p->n=10;
(*q).n=20;
p->show();
(*q).show();
return 0;
}
```

Объявляется класс `MyClass` с целочисленным полем `n` и методом `show()` для отображения значения этого поля. В методе `main()` создаются объекты `a` и `b` класса `MyClass`, а также два указателя `p` и `q` на объекты этого класса (команда `MyClass *p, *q`). Значения указателей определяются командами `p=&a` и `q=&b`, в силу чего переменной-указателю `p` в качестве значения присваивается адрес памяти для объекта `a`, а переменная-указатель `q` получает в качестве значения адрес объекта `b`. Доступ к членам объектов возможен в формате `p->n` или в формате `(*q).n`. В программе использованы оба способа для вызова метода `show()` и определения значения поля `n`. В результате выполнения программы получим:

```
n = 10
n = 20
```

В принципе, работа с указателями на объекты существенно перекликается с методами и синтаксисом использования указателей на структуры.

Существует один особый указатель, который неявно передается каждому методу класса. Это указатель на объект, из которого вызывается метод. Чтобы получить значение указателя на вызывающий метод объект, используют ключевое слово `this`. Пример программного кода, в котором использован указатель `this`, приведен в листинге 8.4.

Листинг 8.4. Использование указателя `this`

```
#include <iostream>
using namespace std;
class MyClass{
public:
    int n;
    int m;
    void show(){
        cout<<"m = "<<this->m<<endl;
        cout<<"n = "<<this->n<<endl;}
    void setmn(int m,int n){
        this->m=m;
        this->n=n;}
    MyClass change(){
        int k;
```

```

        k=m;
        m=n;
        n=k;
        return *this;}
};

int main() {
    MyClass a,b;
    a.setmn(10,20);
    b=a.change();
    a.show();
    b.show();
    return 0;
}

```

У класса `MyClass` есть два поля `m` и `n` типа `int` и три метода. Методом `show()` значения полей объекта выводятся на экран, а методом `change()` меняются значения полей объекта и этот измененный объект возвращается в качестве результата. Методом `setmn()` присваиваются значения полям объекта. Интерес представляют все три метода, в каждом из них использован указатель `this`.

В методе `show()` ссылки на отображаемые поля объекта, из которого вызывается метод, выполнены в виде `this->m` и `this->n`. На самом деле это то же самое, что просто указать имена полей `m` и `n`, что и делалось ранее. Действительно, инструкция `this->m` означает, что необходимо использовать поле `m` того объекта, из которого вызывается метод. Но само по себе имя `m` также означает поле объекта, из которого вызывается метод. Поэтому в методе `show()` вместо означенных инструкций вполне можно было использовать обычные имена полей `m` и `n`. Этого нельзя сказать о методе `setmn()`. Дело в том, что у метода `setmn()` два целочисленных аргумента, и их формальные имена в объявлении метода совпадают с именами полей класса. Поэтому возникает неоднозначность с тем, какая переменная подразумевается при указании имени `m` или `n`. Если более точно, то в теле метода `setmn()` имя `m` или `n` означает соответствующий аргумент метода. Ссылки на одноименные поля теперь с неизбежностью должны выполняться в виде `this->m` и `this->n`. Так, команду `this->m=m` следует понимать как присваивание полю объекта `m` (инструкция `this->m` слева от оператора присваивания) значения аргумента метода `m` (правая часть).

Еще более хитрый код у метода `change()`. Основная часть кода, думается, особых вопросов не вызывает: для объекта, из которого вызван метод, меняются значения полей, т.е. полю `m` присваивается значение поля `n`, и наоборот. Последней командой в теле метода является инструкция `return *this`. Напомним, что `this` является указателем на вызывающий метод объект.

Поэтому `*this` есть ни что иное, как сам этот объект. Он возвращается как результат метода. Таким образом, при вызове метода изменяется вызывавший метод объект (меняются значения его полей) и затем этот измененный объект возвращается в качестве результата. В программе метод `change()` использован для определения значений полей объекта `b` (команда `b=a.change()`). В результате выполнения этой программы получим

```
m = 20
n = 10
m = 20
n = 10
```

Здесь уместно напомнить, что в статических методах, рассматривавшихся в предыдущей главе, указатель `this` в силу достаточно очевидных причин использоваться не может.

Указатели на члены класса

Кроме указателей на объекты класса, в C++ существуют указатели на отдельные члены класса. Указатель на отдельный член существенно отличается от обычного указателя, как по способу объявления, так и по методам использования. **Указатель на член класса задает смещение этого члена. В некотором смысле это относительный адрес члена класса в структуре класса.**

Указатели на члены класса могут ссылаться как на поля, так и на методы. Объявление указателей на поля и на методы выполняется по-разному. При объявлении указателя на поле класса указывается тип для указателя, совпадающий с типом поля, на которое ссылается указатель, а перед именем указателя, кроме обычного в таких случаях оператора `*`, через оператор расширения контекста `::`, указывается и имя класса. Например, если в классе `Класс` объявлено некое поле типа `тип`, то объявление указателя указатель на это поле будет выглядеть так:

```
тип Класс::*указатель;
```

Чтобы впоследствии присвоить значение указателю на член класса, необходимо в правой части после оператора присваивания указать оператор `&`, имя класса и, через оператор расширения контекста `::`, имя поля, на которое ссылается указатель:

```
указатель=&Класс::поле;
```

Ссылка на поле конкретного объекта через указатель на поле класса выполняется в формате `объект.*указатель`.

Более замысловато объявляются указатели на методы класса. Если в классе Класс объявлен метод с аргументами аргументы, возвращающий результат типа тип, то указатель на соответствующий член класса объявляется следующим образом:

```
тип (Класс::*указатель) (аргументы);
```

В отличие от объявления указателя на поле класса, при объявлении указателя на метод класса конструкция Класс::*указатель заключается в круглые скобки, после которых в круглых скобках перечисляется список аргументов, так, как они указывались при описании метода. Значение указателю на метод класса присваивается так же, как и значение указателю на поле класса:

```
указатель=&Класс::метод;
```

Вызов метода конкретного объекта с аргументами аргументы через указатель на метод класса осуществляется в формате (объект.*указатель) (аргументы).

Ранее описывалось, как выполняются обращения к членам объекта через указатель на объект. Допускается совместное использование указателей на объекты и указателей на члены класса. Обращение к члену объекта через указатель на объект выполняется так же, как и обращение через имя объекта, с той лишь разницей, что конструкция объект.меняется на конструкцию указатель->. В листинге 8.5 приведен пример использования указателей на члены класса.

Листинг 8.5. Указатели на члены класса

```
#include <iostream>
using namespace std;
//Объявление класса:
class MyClass{
public:
    int m;
    double n;
    MyClass(int x,double y){
        m=x;
        n=y;}
    void show(bool arg){
        if(arg) cout<<"m = "<<m<<endl;
        else cout<<"n = "<<n<<endl;}
}a(1,2.5),b(3,4.8);
int main(){
    //Объявление указателя на поле класса:
    int MyClass::*p;
```



```

//Объявление обычного указателя:
double *q;
//Объявление указателя на объект:
MyClass *op;
//Объявление указателя на метод класса:
void (MyClass::*s)(bool arg);
//Указателю на поле класса присваивается значение:
p=&MyClass::m;
//Обычному указателю присваивается адрес поля объекта:
q=&a.n;
//Присваивается значение указателю на метод класса:
s=&MyClass::show;
//Присваивается значение указателю на объект:
op=&b;
//Обращение к полю объекта через указатель на поле класса:
cout<<"a.m: "<<a.*p<<endl;
//Обращение к полю объекта через указатель на поле класса
// и указатель на объект:
cout<<"b.m: "<<op->*p<<endl;
//Обращение к полю объекта через обычный указатель:
cout<<"a.n: "<<*q<<endl;
cout<<"a.show():\n";
//Вызов метода объекта через указатель на метод класса:
(a.*s)(true);
cout<<"b.show():\n";
//Вызов метода объекта через указатель на метод класса
// и указатель на объект:
(op->*s)(false);
return 0;
}

```

В результате выполнения программы получим:

```

a.m: 1
b.m: 3
a.n: 2.5
a.show():
m = 1
b.show():
n = 4.8

```

В программе объявлен класс `MyClass` с полем `m` типа `int` и полем `n` типа `double`, а также `void`-метод `show()`, у которого один аргумент типа `bool`. Если аргумент метода равен `true`, методом отображается значение поля `m`. В противном случае полем отображается поле `n`. У класса также есть конструктор с двумя аргументами, определяющими значения полей. При объявлении класса создаются два объекта `a` и `b`.

В главном методе программы командой `int MyClass::*p` объявляется указатель на целочисленное поле класса `MyClass`. В данном случае у класса целочисленное поле всего одно. Тем не менее, пока значение указателю не присвоено, он на это поле не ссылается. Тип в объявлении указателя лишь означает, на какие данные этот указатель может ссылаться. Значение указателю `p` присваивается командой `p=&MyClass::m`. В дальнейшем с помощью этого указателя выполняется обращение к полю `m` объекта `a`, для чего используется команда `a.*p`. В данном случае можно провести аналогию между обычными указателями и указателями на член класса. Если инструкция вида `*указатель` для обычного указателя означает значение переменной или объекта, на которые ссылается указатель, то для указателя на член класса такая инструкция означает имя соответствующего поля.

Следует различать указатель на член класса и указатель на член объекта. Указатель на член объекта – это переменная-указатель, значением которой является адрес поля конкретного объекта. Например, указателю `q`, объявленному как `double *q`, значение присваивается командой `q=&a.n`, в силу чего указатель `q` ссылается на поле `n` объекта `a`. Выполнить с помощью этого указателя обращение к полю `n` другого объекта не получится.

Командой `void (MyClass::*s)(bool arg)` объявляется указатель на метод класса. Значение указателю присваиваем командой `s=&MyClass::show`. Указатель `s` использован в командах `(a.*s)(true)` и `(b.*s)(false)` для вызова метода `show()` объектов `a` и `b` соответственно.

Обращение к полям и методам объектов могут выполняться через указатель на объект и указатель на член класса. Так, командой `MyClass *op` объявляется указатель `op` на объект класса `MyClass`. Командой `op=&b` определяется значение указателя (в качестве значения указатель получает адрес объекта `b`). С помощью этого указателя в командах `op->*p` и `(op->*s)(false)` выполняется обращение к членам объекта `b`.

Использование ссылок на объекты

Не копируйте человека, если вы не способны ему подражать.

Й. Берра

На объекты, как и на переменные базовых типов, можно выполнять ссылки. **Ссылка – это фактически псевдоним объекта или переменной.** Существует несколько основных способов использования ссылок. Рассмотрим их.

Независимая ссылка на объект позволяет использовать для одного и того же объекта разные названия. Ссылка при объявлении сразу инициализирует-

ся. В качестве значения ссылки указывается объект, для которого создается ссылка. Перед именем ссылки в объявлении указывается оператор `&`. В качестве типа ссылки указывают имя класса, на объект которого выполняется ссылка. После создания ссылки к объекту можно обращаться через его имя или через имя ссылки на этот объект. Пример приведен в листинге 8.6.

Листинг 8.6. Независимая ссылка на объект

```
#include <iostream>
using namespace std;
//Объявление класса:
class MyClass{
    public:
        double x;
};
int main(){
    //Объект класса:
    MyClass obj;
    //Ссылка на объект:
    MyClass &ref=obj;
    //Обращение к объекту по имени и через ссылку:
    obj.x=10;
    cout<<"x = "<<ref.x<<endl;
    ref.x=100;
    cout<<"x = "<<obj.x<<endl;
    return 0;
}
```

В результате выполнения этой программы получим:

```
x = 10
x = 100
```

В программе объявлен класс `MyClass`, у которого всего одно поле `x` типа `double`. В главном методе программы командой `MyClass obj` объявляется объект `obj`. Следующей командой `MyClass &ref=obj` создается независимая ссылка `ref` на объект `obj`. После этого к объекту можно обращаться либо через имя `obj`, либо через имя `ref`. Например, если присвоить значение полю объекта командой `obj.x=10`, то у выражения `ref.x` значение будет то же самое. И наоборот, после выполнения команды `ref.x=100` соответствующим образом изменится и результат выражения `obj.x`.

Как и с базовыми типами, объекты в аргументах функции можно передавать по значению и по ссылке. По умолчанию объекты передаются по значению. Для передачи аргумента по ссылке перед именем аргумента указывается оператор `&`. Пример передачи аргумента-объекта по ссылке и по значению приведен в листинге 8.7.

Листинг 8.7. Передача аргументов по ссылке и по значению

```
#include <iostream>
using namespace std;
class MyClass{
public:
    double x;
};
//Передача аргумента по значению:
double f1(MyClass obj){
    obj.x*=10;
    return obj.x;
}
//Передача аргумента по ссылке:
double f2(MyClass &obj){
    obj.x*=10;
    return obj.x;
}
int main(){
    MyClass obj;
    obj.x=5;
    cout<<"x = "<<f1(obj)<<endl;
    cout<<"x = "<<obj.x<<endl;
    cout<<"x = "<<f2(obj)<<endl;
    cout<<"x = "<<obj.x<<endl;
    return 0;
}
```

В результате получим:

```
x = 50
x = 5
x = 50
x = 50
```

Как и ранее, класс `MyClass` содержит `double`-поле `x`. В программе также объявлены две функции `f1()` и `f2()`, возвращающие значение типа `double`, а аргумент – объект класса `MyClass`. В функции поле объекта-аргумента умножается на 10, после чего полученное значение возвращается в качестве результата функции. Разница между функциями заключается в способе передачи аргумента: для функции `f1()` передается по значению, а для функции `f2()` – по ссылке. В первом случае хотя функцией результат вычисляется правильный, значение поля объекта, переданного аргументом функции, не меняется (напомним, что при передаче аргумента по значению в функции обрабатывается копия реального аргумента). Во втором случае, при передаче аргумента по ссылке, поле объекта, переданного аргументом функции, меняется.

Ссылка на объект не только может передаваться аргументом функции, но и возвращаться функцией в качестве результата. При объявлении такой функции перед ее именем указывается оператор `&`. Пример использования функции, возвращающей в качестве результата ссылку на объект, приведен в листинге 8.8.

Листинг 8.8. Результат функции – ссылка на объект

```
#include <iostream>
using namespace std;
//Класс и два объекта:
class MyClass{
public:
    double x;
    void show(){
        cout<<"x = "<<x<<endl;}
}a,b;
//Результатом функции является ссылка:
MyClass &f(bool arg){
    if(arg) return a;
    else return b;}
int main(){
    //Обращение к объектам через функцию-ссылку:
    f(true).x=5;
    f(true).show();
    //Проверка результата:
    cout<<"x = "<<a.x<<endl;
    //Обращение к объектам через функцию-ссылку:
    f(false).x=10;
    f(false).show();
    //Проверка результата:
    cout<<"x = "<<b.x<<endl;
    return 0;
}
```

Результат выполнения программы имеет вид:

```
x = 5
x = 5
x = 10
x = 10
```

Класс `MyClass` имеет поле `x` типа `double` и метод `show()`, которым отображается значение этого поля. В программе также определяется функция `f()` (прототип функции `MyClass &f(bool arg)`) – перед именем функции указывается оператор `&`), у которой логический аргумент, а результатом является ссылка на один из объектов `a` или `b` в зависимости от значения аргумента функции: при значении аргумента `true` функцией возвращается ссылка на объект `a`, и в противном случае, при значении аргумента `false`, возвращается ссылка на объект `b`.

Обращение к объектам `a` и `b` может осуществляться обычным способом через имя объектов или через функцию-ссылку: `f(true)` является обращением к объекту `a` и `f(false)` является обращением к объекту `b`. Например, обратиться к полю `x` объекта `a` можно командой `f(true).x`, а вызвать метод `show()` объекта `b` можно командой `f(false).show()`.

Массивы объектов

Ну кто так строит, кто так строит?

Из к/ф «Чародеи»

Массивы объектов создаются и используются точно так же, как и массивы элементов базовых типов, только в качестве типа элементов массива указываются имя соответствующего класса. Пример объявления и использования массива объектов приведен в листинге 8.9.

Листинг 8.9. Массив объектов

```
#include <iostream>
using namespace std;
class MyClass{
public:
    double x;
    void show(){
        cout<<"x = "<<x<<endl;}
};
int main(){
    const n=5;
    int i;
    MyClass objs[n];
    for(i=0;i<n;i++){
        objs[i].x=2*i+1;
        cout<<i+1<<": ";
        objs[i].show();
    }
    return 0;
}
```

При выполнении программы получим следующий результат:

```
1: x = 1
2: x = 3
3: x = 5
4: x = 7
5: x = 9
```

В программе описывается класс `MyClass` с `double`-полем `x` и методом `show()` для отображения значения поля. В методе `main()` командой `MyClass objs[n]` объявляется массив `objs` из `n` объектов класса `MyClass`. Обращение к элементам массива в рамках оператора цикла осуществляется в формате `objs[i]`: к полю `x` объектов-элементов массива обращение имеет вид `objs[i].x`, а для вызова метода `show()` используется команда вида `objs[i].show()`.

Если в классе описан конструктор с обязательной передачей аргументов, то при объявлении массива его необходимо сразу инициализировать. Для инициализации в фигурных скобках перечисляются вызовы конструкторов с нужными аргументами. Пример приведен в листинге 8.10.

Листинг 8.10. Инициализация массива объектов

```
#include <iostream>
using namespace std;
class MyClass{
public:
    double x;
    void show(){
        cout<<"x = "<<x<<endl;}
    MyClass(double z){
        x=z;}
};
int main(){
    int i;
    MyClass objs[]={MyClass(1),MyClass(3),MyClass(5)};
    for(i=0;i<3;i++){
        cout<<i+1<<" : ";
        objs[i].show();}
    return 0;
}
```

Интерес представляет объявление с одновременной инициализацией массива `MyClass objs[]={MyClass(1),MyClass(3),MyClass(5)}`. В результате создается массив из трех объектов, значения полей которых равны соответственно 1, 3 и 5. В результате выполнения программы получим

```
1: x = 1
2: x = 3
3: x = 5
```

Отметим, что если у конструктора один аргумент, то в списке инициализации достаточно перечислить аргументы конструктора. Другими словами, в рассмотренном примере при инициализации массива можно было использовать команду `MyClass objs[]={1,3,5}`.

Динамическое выделение памяти под объекты

Делить будем по-честному: это тебе, это снова тебе, это опять тебе, это все время тебе. Так, я себя не обделил?

Из к/ф «Свадьба в Малиновке»

Для динамического выделения памяти под объекты класса используют оператор `new`. Сначала объявляется указатель на объект соответствующего класса, после чего в формате `указатель=new класс` соответствующей командой под объект выделяется место в памяти и адрес передается указателю. Если при создании объекта конструктору (конструкторам посвящена глава 9) необходимо передать аргументы, они указываются в круглых скобках после имени класса. Для удаления объекта из памяти используется команда `delete указатель`, где указатель является указателем на удаляемый объект. Примеры динамического выделения памяти под объекты приведены в листинге 8.11.

Листинг 8.11. Динамическое выделение памяти под объект

```
#include <iostream>
using namespace std;
class MyClass{
public:
double x;
void show(){
    cout<<"x = "<<x<<endl;}
MyClass(double z){
    x=z;
    cout<<"Object with x = "<<x<<" has been created!\n";}
MyClass(){
    x=0;
    cout<<"Object with x = "<<x<<" has been created!\n";
}
~MyClass(){
    cout<<"Object with x = "<<x<<" has been deleted!\n";
}
};

int main(){
    MyClass *p;
    p=new MyClass;
    p->show();
```



```

delete p;
p=new MyClass(1);
p->show();
delete p;
return 0;
}

```

В классе `MyClass` описан конструктор с одним аргументом и без аргументов, а также деструктор (конструктор – метод, вызываемый при создании объекта, деструктор вызывается при удалении объекта из памяти). Подробнее конструкторы и деструкторы обсуждаются в следующей главе. При динамическом выделении памяти с помощью команды `p=new MyClass` полю `x` создаваемого объекта присваивается значение 0 (вызывается конструктор без аргументов). Освобождение памяти, занятой объектом, осуществляется командой `delete p`. При динамическом выделении памяти под объект командой `p=new MyClass(1)` вызывается вариант конструктора с аргументом и полю `x` присваивается значение 1. В результате получаем:

```

Object with x = 0 has been created!
x = 0
Object with x = 0 has been deleted!
Object with x = 1 has been created!
x = 1
Object with x = 1 has been deleted!

```

Динамические массивы объектов создаются аналогично динамическим массивам базовых типов.

Дружественные функции и классы

Нет друзей у того, у кого их много.

Аристотель

Как отмечалось ранее, члены класса могут быть закрытыми и открытыми. К закрытым членам класса доступ существует только внутри класса. Однако нередко случается необходимость, чтобы член класса оставался закрытым, но некоторые внешние функции или методы имели к нему доступ. Такие **внешние функции, которые имеют доступ к закрытым членам класса, называются дружественными функциями.**

Чтобы задекларировать функцию как дружественную для класса, необходимо указать прототип этой функции в описании класса, предварив его ключевым словом `friend`. Пример использования дружественной функции приведен в листинге 8.12.

Листинг 8.12. Использование дружественной функции

```

#include <iostream>
using namespace std;
//Класс с закрытым полем:
class MyClass{
    double x;
public:
    MyClass(double z){x=z;}
    //Дружественная функция:
    friend void show(MyClass obj);
};
//Описание дружественной функции:
void show(MyClass obj){
    cout<<"x = "<<obj.x<<endl;}
int main(){
    MyClass a(10);
    //Дружественная функция имеет доступ к закрытым членам:
    show(a);
    return 0;
}

```

В данном случае поле `x` класса `MyClass` является закрытым, поэтому формально получить доступ к этому полю класса можно только внутри класса. Для заполнения этого поля предусмотрен конструктор, которому передается один аргумент. Однако больше никаких методов для получения доступа в классе нет. Отображение значения поля `x` будет осуществляться с помощью внешней функции `show()`. Чтобы функция имела доступ к закрытому полю, она объявлена в классе как дружественная, для чего в описание класса добавлена инструкция `friend void show(MyClass obj)`. Благодаря тому, что функция дружественная к классу, корректной является команда `show(a)`, которой отображается значение закрытого поля.

Обращаем внимание читателя, что, хотя дружественная функция не является членом класса, ее прототип в классе указан.

Концепция дружественных функций может показаться несколько искусственной, но это не так. Представим себе ситуацию, когда есть два класса с закрытыми полями и необходимо получить доступ к полям обоих классов. Это один из тех случаев, когда целесообразным является применение дружественных функций. Пример приведен в листинге 8.13.

Листинг 8.13. Дружественные функции

```

#include <iostream>
using namespace std;

```

```

//Анонс класса:
class B;
//Класс с закрытым полем:
class A{
    double x;
    public:
    A(double z){x=z;}
    //Дружественная функция с двумя аргументами:
    friend double summa(A a,B b);
}a(3.5);
//Класс с закрытым полем:
class B{
    double y;
    public:
    B(double z){y=z;}
    //Дружественная функция с двумя аргументами:
    friend double summa(A a,B b);
}b(2.3);
double summa(A a,B b){
    return a.x+b.y;
}
int main(){
    //Вызов дружественной функции:
    cout<<"Total is "<<summa(a,b)<<endl;
    return 0;
}

```

В программе два класса (А и В), у каждого из которых есть закрытое поле типа double. Функцией `summa()` вычисляется сумма закрытых полей объектов, переданных ей в качестве аргументов. Первый аргумент функции – объект класса А, второй аргумент функции – объект класса В. Чтобы функция имела доступ к полям обоих классов, в каждом классе эта функция объявлена как дружественная. Кроме того, поскольку впервые прототип дружественной функции в программном коде появляется до описания класса В, до этого места в программе приведен анонс класса В (команда `class B` перед объявлением класса А). Такой анонс необходим для того, чтобы идентифицировать как класс объявление типа во втором аргументе функции.

В результате выполнения программы получим

```
Total is 5.8
```

Дружественными по отношению к классу могут быть отдельные методы другого класса или целый класс. В последнем случае все методы дружественного класса имеют доступ к закрытым полям и методам исходного

класса. Листинг 8.14 содержит пример использования дружественного метода.

Листинг 8.14. Дружественные методы

```
#include <iostream>
using namespace std;
//Анонс класса:
class B;
//Класс с закрытым полем и методом:
class A{
    double x;
public:
    A(double z){x=z;}
    double summa(B b);
}a(3.5);
//Класс с закрытым полем и дружественным методом:
class B{
    double y;
public:
    B(double z){y=z;}
    //Дружественный метод:
    friend double A::summa(B b);
}b(2.3);
int main(){
    //Вызов дружественного метода:
    cout<<"Total is "<<a.summa(b)<<endl;
    return 0;
}
double A::summa(B b){
    return x+b.y;}
```

Функциональность программы по сравнению с предыдущим случаем не изменилась, но теперь сумма полей классов вычисляется с помощью метода класса A.

У метода `summa()` класса A уже один аргумент (объект класса B) – второй объект методу передавать не нужно, поскольку из этого объекта вызывается метод и доступ к объекту у метода имеется автоматически.

В классе B инструкцией `friend double A::summa(B b)` метод `summa()` класса A объявлен как дружественный. Обращаем внимание на два обстоятельства. Во-первых, при объявлении дружественного метода явно указано, к какому классу он принадлежит (перед именем метода указано название класса). Во-вторых, описание метода `summa()` должно осуществляться только после того, как описаны классы, объекты которых используются в методе – в данном случае это класс B.

В листинге 8.15 программный код несколько изменен так, чтобы один класс был дружественным к другому классу. При этом в дружественный класс добавлен еще один метод `product()` для вычисления произведения закрытых полей классов.

Листинг 8.15. Дружественные классы

```
#include <iostream>
using namespace std;
class B;
class A{
    double x;
public:
    A(double z){x=z;}
    double summa(B b);
    double product(B b);
}a(3.5);
class B{
    double y;
public:
    B(double z){y=z;}
    friend class A;
}b(2.3);
int main(){
    cout<<"Total is "<<a.summa(b)<<endl;
    cout<<"Product is "<<a.product(b)<<endl;
    return 0;
}
double A::summa(B b){
    return x+b.y;}
double A::product(B b){
    return x*b.y;}
```

В классе `B` имеется инструкция `friend class A`, которой класс `A` объявляется как дружественный к классу `B`. В этом случае все методы класса `A` имеют доступ к членам класса `B`, в том числе и закрытым. Поэтому вполне корректными являются команды `a.summa(b)` и `a.product(b)` в главном методе программы. В результате выполнения программы получаем

```
Total is 5.8
Product is 8.05
```

Следует отметить, что дружественные классы используются не очень часто – существуют более эффективные способы реализации доступа методов одного класса к членам другого, главным среди которых является механизм наследования.

Примеры решения задач

Ваша опера мне понравилась. Пожалуй, я напишу к ней музыку.

Л. ван Бетховен

Приведенные далее примеры решения задач иллюстрируют расширенные методы работы с классами и объектами.

■ Комплексная экспонента

Напишем программу для вычисления экспоненты от комплексного аргумента. Комплексные числа реализуем с помощью пользовательского класса. При вычислении экспонент от комплексного аргумента $z = x + iy$ воспользуемся тем, что $\exp(z) = \exp(x)(\cos(y) + i \sin(y))$. Таким образом, результатом вычисления экспоненты от комплексного числа является комплексное число. Поэтому аргументом соответствующей функции должен быть объект класса, через который реализуются комплексные числа, а в качестве результата функцией возвращается объект того же класса. Программный код приведен в листинге 8.16.

Листинг 8.16. Комплексная экспонента

```
#include <iostream>
#include <cmath>
using namespace std;
//Класс для реализации комплексных чисел:
class Complex{
public:
double Re, Im;
//Метод для отображения комплексного числа:
void show(){
cout<<Re;
if(Im>0) cout<<"+"<<Im<<"i"<<endl;
if(Im<0) cout<<"-"<<-Im<<"i"<<endl;}
};
//Экспонента от комплексного числа:
Complex cExp(Complex z){
Complex tmp;
tmp.Re=exp(z.Re)*cos(z.Im);
tmp.Im=exp(z.Re)*sin(z.Im);
return tmp;
}
```

```
int main(){
    Complex z;
    cout<<"Re: ";
    cin>>z.Re;
    cout<<"Im: ";
    cin>>z.Im;
    cout<<"z=";
    z.show();
    cout<<"exp(z)=";
    cExp(z).show();
    return 0;}
```

Результат выполнения программы может иметь следующий вид (ввод пользователя выделен жирным шрифтом):

```
Re: 1
Im: 2
z=1+2i
exp(z)=-1.1312+2.47173i
```

Программный код достаточно прост: класс `Complex` для реализации комплексных чисел имеет два поля и метод для отображения символьного представления для комплексного числа. Функция `cExp()` для вычисления экспоненты от комплексного аргумента в качестве последнего получает объект класса `Complex`. Результатом функции также является объект класса `Complex`. Для вычисления экспоненты, косинуса и синуса от обычных функций используются встроенные математические функции C++.

■ Динамический список

Напишем программу с реализацией динамического массива объектов. Похожая задача рассматривалась в главе 6, посвященной структурам. Здесь базовыми элементами динамического списка будут объекты специально созданного класса, одно из полей которых является указателем на объект того же класса. Формирование списка реализуется через цепочку последовательных ссылок. Для каждого объекта из списка память выделяется динамически. Программный код с реализацией списка приведен в листинге 8.17.

Листинг 8.17. Динамический список

```
#include <iostream>
using namespace std;
//Класс для объектов-элементов списка:
class Numbers{
```

```

    public:
    int n;
    Numbers *p;
    };

//Функция для создания списка:
Numbers *make(int N){
    Numbers *p1,*p2;
    int i;
    p1=new Numbers;
    p1->n=1;
    p1->p=NULL;
    for(i=2;i<=N;i++){
        p2=new Numbers;
        p2->n=i;
        p2->p=p1;
        p1=p2;}
    return p1;}

//Функция для отображения элементов списка:
void showAll(Numbers *q){
    do{
        cout<<q->n<<" : "<<q<<endl;
        q=q->p;
    }while(q!=NULL);
    }

//Функция для удаления списка из памяти:
void deleteAll(Numbers *q){
    Numbers *q1;
    do{
        q1=q;
        cout<<"deleted: "<<q<<endl;
        q=q1->p;
        delete q1;
    }while(q!=NULL);
    }

int main(){
    int n;
    Numbers *q;
    cout<<"Enter n = ";
    cin>>n;
    q=make(n);
    showAll(q);
    deleteAll(q);
    return 0;}

```


Класс `Numbers`, через который реализуются объекты-элементы списка, достаточно прост. У него всего два поля: целочисленное поле `n` и поле-указатель `p`, через которое реализуется связь между элементами списка.

Динамический список создается с помощью функции `make()`, которой в качестве результата возвращается указатель на последний элемент списка. Аргументом функции передается целочисленная переменная, определяющая количество элементов в списке. Функцией последовательно выделяется память для новых объектов-элементов. Поля `n` объектов нумеруются натуральными числами в порядке создания объектов, а поле-указатель `p` ссылается на созданный на предыдущем шаге элемент (объект). Поле `p` первого созданного объекта списка получает в качестве значения нулевой указатель (значение `NULL`).

Для отображения элементов списка используется функция `showAll()`. Аргументом функции передается указатель на последний элемент списка, а элементы отображаются в формате «*значение поля n : адрес элемента*». Для удаления элементов списка из памяти предлагается функция `deleteAll()`. В качестве аргумента функции необходимо передать указатель на последний объект-элемент списка. При удалении объекта из памяти на экран выводится соответствующее сообщение с адресом удаленного объекта. Сразу отметим, что такого рода действия лучше реализовывать через деструкторы.

В результате выполнения программы можем получить нечто следующее (жирным шрифтом выделен ввод пользователя):

```
Enter n = 5
5 : 00322AA8
4 : 00322A70
3 : 00321148
2 : 00321110
1 : 003210D8
deleted: 00322AA8
deleted: 00322A70
deleted: 00321148
deleted: 00321110
deleted: 003210D8
```

Обращаем внимание читателя, что как для функции `showAll()`, так и для функции `deleteAll()` принципиально важно, чтобы последний элемент списка содержал нулевой указатель – именно это обстоятельство выбрано в качестве критерия остановки итерационного процесса.

■ Векторное произведение

Реализуем с помощью классов процесс вычисления векторного произведения двух векторов. Напомним, что векторным произведением векторов $\vec{a} = (a_1, a_2, a_3)$ и $\vec{b} = (b_1, b_2, b_3)$ называется вектор $\vec{c} = [\vec{a} \times \vec{b}] = (a_2b_3 - b_2a_3, a_3b_1 - b_3a_1, a_1b_2 - b_1a_2)$. Вектор реализуем в виде объекта класса, координаты вектора определим как поля класса, а векторное произведение будем вычислять с помощью метода класса. Программный код приведен в листинге 8.18.

Листинг 8.18. Векторное произведение

```
#include <iostream>
using namespace std;
class Vector{
public:
    double x,y,z;
    Vector vprod(Vector obj){
        Vector tmp;
        tmp.x=y*obj.z-obj.y*z;
        tmp.y=z*obj.x-obj.z*x;
        tmp.z=x*obj.y-obj.x*y;
        return tmp;
    }
    void show(){
        cout<<"("<<x<<" "<<y<<" "<<z<<"\n";
    }
};
int main(){
    Vector a,b,c;
    a.x=1;
    a.y=0;
    a.z=0;
    b.x=0;
    b.y=1;
    b.z=0;
    c=a.vprod(b);
    c.show();
    b.vprod(a).show();
    return 0;}
```

В результате выполнения программы получаем такой результат:

```
(0,0,1)
(0,0,-1)
```

Для удобства в классе `Vector`, кроме полей `x`, `y` и `z` и метода `vprod()` для вычисления векторного произведения, описан еще и метод `show()`, с помощью которого отображаются координаты вектора. Обращаем внимание чи-

тателя на два способа вызова этого метода в главном методе программы. В первом случае (команда `c.show()`) метод вызывается из объекта `c`. Во втором случае (команда `b.vprod(a).show()`) метод вызывается из временно-го объекта `b.vprod(a)` – результата вызова функции `vprod()` из объекта `b`.

■ Интерполяционный полином

Задача по интерполяции данных формулируется следующим образом. Имеется набор узловых точек $\{x_i\}$ и значений некоторой функции в этих точках $\{y_i\}$ соответственно, $i = 0, 1, \dots, n$. Необходимо построить такой полином $L_n(x)$ степени n , чтобы он проходил через все узловые точки, т.е. чтобы для всех $i = 0, 1, \dots, n$ выполнялось равенство $L_n(x_i) = y_i$.

Один из вариантов решения – интерполяционный полином Лагранжа $L_n(x) = \sum_{i=0}^n y_i \varphi_i(x)$, где функции $\varphi_i(x) = \prod_{\substack{j=0, \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}$ таковы, что

$\varphi_i(x_j) = \delta_{ij}$, а δ_{ij} – символ Кронекера. Именно интерполяционный полином Лагранжа реализуем в виде метода класса для выполнения интерполяции на основе данных, представленных в виде массивов-полей того же класса. Программный код приведен в листинге 8.19.

Листинг 8.19. Интерполяционный полином

```
#include <iostream>
#include <cstdlib>
using namespace std;
const int n=9;
class Interpol{
    double psi(int i,double z){
        double tmp=1;
        int j;
        for(j=0;j<i;j++) tmp*=(z-x[j])/(x[i]-x[j]);
        for(j=i+1;j<=n;j++) tmp*=(z-x[j])/(x[i]-x[j]);
        return tmp;
    }
public:
    double x[n+1];
    double y[n+1];
    double Lagr(double z){
        double s=0;
        int i;
        for(i=0;i<=n;i++) s+=y[i]*psi(i,z);
        return s;
    }
};
```

```

int main() {
    int i;
    double x;
    Interpol obj;
    for(i=0; i<=n; i++) {
        obj.x[i]=i;
        obj.y[i]=i*(i-n/3)*(i-n+1)+0.001*(rand()%100-50);
        cout<<obj.y[i]<<" ";
    }
    cout<<endl;
    do{
        cout<<"x = ";
        cin>>x;
        cout<<"L(x)="<<obj.Lagr(x)<<endl;
    }while(x>=0&& x<=n);
    return 0;}

```

В программе описан класс с двумя полями-массивами (узловые точки и значения функции в этих узловых точках), закрытый метод `psi()` для вычисления функций $\varphi_i(x)$, а также метод `Lagr()` для вычисления интерполяционного полинома. В главном методе программы создается соответствующий объект, узловые точки определяются целыми числами, значения функции определяются на основе полиномиальной зависимости третьей степени со случайной поправкой. Значения функции в узловых точках выводятся на экран, после чего в рамках оператора цикла пользователю предлагается ввести значение аргумента, для которого вычисляется значение интерполяционного полинома. Цикл продолжается до тех пор, пока пользователь не введет значение, выходящее за пределы диапазона узловых точек. Результат выполнения программы может выглядеть так (жирным шрифтом выделен вод пользователя):

```

-0.009 14.017 11.984 -0.05 -15.981 -30.026 -35.972 -27.992
0.012 54.014
x = 0
L(x)=-0.009
x = 3
L(x)=-0.05
x = 9
L(x)=54.014
x = 5.5
L(x)=-34.3899
x = 9.3
L(x)=74.5793

```

Видим, в узловых точках значения функции и интерполяционного полинома совпадают, как и должно быть.

■ Производная для полинома

Напишем программу, с помощью которой, во-первых, реализуем в виде объекта класса зависимость полиномиального типа, а во-вторых, предусмотрим механизм вычисления производной для соответствующего полинома. Соответствующий программный код приведен в листинге 8.20.

Листинг 8.20. Производная для полинома

```
#include <iostream>
#include <cmath>
using namespace std;
//Степень полинома:
const int n=4;
//Класс для реализации полинома:
class Polynom{
public:
    //Коэффициенты полинома в точке:
    double a[n+1];
    //Значение полинома:
    double P(double x){
        double s=0;
        int i;
        for(i=0;i<=n;i++) s+=a[i]*pow(x,i);
        return s;
    }
    //Производная:
    Polynom Deriv(){
        Polynom tmp;
        int i;
        tmp.a[n]=0;
        for(i=n-1;i>=0;i--) tmp.a[i]=a[i+1]*(i+1);
        return tmp;
    }
};
//Отображение таблицы значений:
void show(Polynom obj,double x1,double x2){
    double h=(x2-x1)/5;
    int i;
    cout<<"x: ";
    for(i=0;i<=5;i++)
        printf("%12f",x1+i*h);
    cout<<endl<<"P(x): ";
    for(i=0;i<=5;i++)
        printf("%12f",obj.P(x1+i*h));
    cout<<endl;
}

int main() {
```

```
//Полином и его производные:
Polynom Q1,Q2,Q3;
int i;
for(i=0;i<=n;i++) Q1.a[i]=1;
//Первая производная:
Q2=Q1.Deriv();
//Вторая производная:
Q3=Q2.Deriv();
show(Q1,0,1);
show(Q2,0,1);
show(Q3,0,1);
return 0;}
```

В классе `Polynom` определено поле-массив `a` для коэффициентов массива, метод `P()` для вычисления значения полинома в точке, а также метод `Deriv()` для вычисления производной. Этим методом создается новый объект класса `Polynom`, коэффициенты которого вычисляются так, чтобы объект соответствовал производной от исходного полинома. Отдельно описана функция `show()` для отображения в виде таблицы значений полинома. Аргументами функции передаются объект, соответствующий табулируемому полиному, а также диапазон изменений аргумента полинома.

В главном методе программы создается три объекта класса `Polynom`: исходный полином и две его производных. Результат выполнения программы имеет вид:

```
x: 0.000000 0.200000 0.400000 0.600000 0.800000 1.000000
P(x): 1.000000 1.249600 1.649600 2.305600 3.361600 5.000000
x: 0.000000 0.200000 0.400000 0.600000 0.800000 1.000000
P(x): 1.000000 1.552000 2.536000 4.144000 6.568000 10.000000
x: 0.000000 0.200000 0.400000 0.600000 0.800000 1.000000
P(x): 2.000000 3.680000 6.320000 9.920000 14.480000 20.000000
```

В качестве исходного полинома использовалось выражение $1 + x + x^2 + x^3 + x^4$. Желающие могут проверить, что производные полинома вычисляются корректно.

■ Матричная экспонента

Если подразумевать под экспонентой ряд вида $\exp(z) = \sum_{n=0}^{\infty} \frac{z^n}{n!}$, можно вычислять экспоненту и для матричного аргумента. Если A – квадратная матрица, то матричная экспонента определяется как $\exp(A) = E + A + \frac{1}{2!}A^2 + \frac{1}{3!}A^3 + \dots + \frac{1}{n!}A^n + \dots$, где E – единичная

матрица того же ранга, что и матрица A . Рассмотрим программный код, представленный в листинге 8.21, в котором реализуется процесс вычисления матричной экспоненты для квадратных матриц ранга 2. Матрицы реализуются в виде объектов класса.

Листинг 8.21. Матричная экспонента

```
#include <iostream>
using namespace std;
//Количество членов ряда:
const int N=100;
//Класс для реализации матриц:
class Matrix{
public:
double a[2][2];
//Метод для умножения матриц:
Matrix mult(Matrix obj){
    Matrix T;
    int i,j,k;
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            T.a[i][j]=0;
            for(k=0;k<2;k++){
                T.a[i][j]+=a[i][k]*obj.a[k][j];
            }
        }
    }
    return T;}
//Метод для суммирования матриц:
Matrix mSum(Matrix obj){
    Matrix T;
    int i,j;
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            T.a[i][j]=a[i][j]+obj.a[i][j];
        }
    }
    return T;}
//Деление матрицы на число:
Matrix div(double x){
    int i,j;
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            a[i][j]/=x;
        }
    }
    return *this;}
//Отображение матрицы:
void mShow(){
    printf("%8f%s%8f%s",a[0][0]," ",a[0][1],"\\n");
    printf("%8f%s%8f%s",a[1][0]," ",a[1][1],"\\n");
};
//Матричная экспонента:
Matrix mExp(Matrix obj){
    int i;
```

```

Matrix E, T;
E.a[0][0]=1;
E.a[1][1]=1;
E.a[0][1]=0;
E.a[1][0]=0;
T=E;
for(i=1;i<=N;i++){
    T=T.mult(obj);
    T=T.div(i);
    E=E.mSum(T); }
return E;}

int main() {
Matrix A;
A.a[0][0]=1;
A.a[0][1]=-1;
A.a[1][0]=2;
A.a[1][1]=1;
A.mShow();
cout<<endl;
mExp(A).mShow();
return 0;}

```

В программе объявляется класс `Matrix`, через который реализуются матрицы. Элементы матрицы представлены как поле-массив `a`. Для умножения двух матриц используется метод `mult()`. Для суммирования матриц используется метод `mSum()`, а деление матрицы на число выполняется с помощью метода `div()`. Все эти операции придется выполнять при вычислении матричной экспоненты. Кроме этого, в классе объявлен метод `mShow()` для отображения элементов матрицы.

Внешняя функция `mExp()` возвращает в качестве результата (матричная экспонента) объект класса `Matrix`. Объект того же класса передается функции в качестве аргумента. В этой функции вызываются методы класса `Matrix`. Вообще же в таких случаях вместо использования методов разумнее перегружать соответствующие операторы (перегрузке операторов посвящена глава 10).

В главном методе программы создается объект класса `Matrix`, заполняются его поля, после чего вычисляется матричная экспонента. В результате выполнения программы получаем:

```

1.000000 -1.000000
2.000000 1.000000
0.423899 -1.898600
3.797200 0.423899

```

Аналогичным способом могут вычисляться и другие функции для матричных аргументов.

Резюме

На самом деле я не говорил ничего из того, что я говорил.

Й. Берра

1. Объекты, как и переменные базовых типов, могут передаваться аргументами функциям и методам. При этом в качестве типа переменной-объекта указывается имя класса, к которому принадлежит соответствующий объект.
2. Объекты могут возвращаться функциями как результат. В качестве типа результата функции указывается имя класса, объект которого возвращается в качестве результата функции или метода.
3. Для объектов могут создаваться указатели. Значением переменной-указателя является адрес первой ячейки области памяти, выделенной под объект.
4. Кроме указателей на объекты класса, в C++ существуют указатели на отдельные члены класса. Указатель на член класса задает смещение этого члена – относительный адрес члена класса в структуре класса. Указатели на члены класса могут ссылаться как на поля, так и на методы.
5. На объекты, как и на переменные базовых типов, можно выполнять ссылки. Ссылка – это псевдоним объекта или переменной.
6. Элементами массива могут быть объекты. Массивы объектов создаются и используются точно так же, как и массивы элементов базовых типов, только в качестве типа элементов массива указывается имя соответствующего класса.
7. Под объекты может динамически выделяться память. Делается это так же, как и в случае обычных переменных. Для динамического выделения памяти под объекты класса используют оператор `new`. Освобождается память с помощью команды `delete`.
8. В C++ допускается создание внешних функций, которые имеют доступ к закрытым членам класса. Такие функции называются дружественными. Для объявления функции как дружественной к классу в этом классе указывается прототип функции с ключевым словом `friend`. Как дружественные могут объявляться и целые классы.

Контрольные вопросы

Об уме человека вернее судить по его вопросам, нежели по его ответам.

Ж. Левис

1. Каким образом объекты передаются аргументами функциям и методам?
2. Каков механизм возвращения функцией или методом объекта в качестве результата?
3. Как создаются указатели на объект?
4. Что такое указатель на член класса и как он создается?
5. Как создается ссылка на объект и как она используется?
6. Как создаются массивы объектов?
7. Как под объекты динамически выделяется память и как память освобождается?
8. Что такое дружественные функции, как они объявляются и используются?

Задачи для самостоятельного решения

В предлагаемых для самостоятельного решения задачах необходимо создать специальные классы. Для справки можно обратиться к разделу, в котором рассматривались примеры решения задач. Аналогичные задачи рассматривались в главе 6, посвященной структурам. В данном случае вместо структур необходимо использовать классы.

Задача 1. Написать программу для создания бинарного дерева, элементами которого являются объекты класса (см. примеры из главы 6).

Задача 2. Написать программу для создания дерева объектов, каждый из которых ссылается на два других объекта. Ссылка осуществляется через поля-указатели. Объекты организуются по следующему принципу. Начальный (первый) объект ссылается на два объекта (второй и третий), каждый из которых ссылается на четвертый общий объект и друг на друга. Четвертый объект ссылается на два объекта и т.д.

Задача 3. Написать программу для создания дерева объектов, каждый из которых ссылается на два других объекта. Ссылка осуществляется через поля-указатели. Начальный (первый) объект ссылается на два объекта (второй и третий), каждый из которых ссылается друг на друга и на еще один объект (четвертый и пятый). Четвертый и пятый объекты ссылаются друг на друга и на один общий объект (шестой), который ссылается на два объекта, и т.д.

Задача 4. Написать программу для создания триарного дерева объектов. В такой структуре каждый объект ссылается на три объекта такого же типа. Каждый из этих объектов, в свою очередь, ссылается на три объекта и т.д.

Задача 5. Написать программу для создания N-кратного дерева объектов. В такой структуре каждый объект ссылается на N объектов такого же типа. Каждый из этих объектов, в свою очередь, ссылается на N объектов и т.д. Параметр N задается как константа. Поля-указатели объекта реализовать в виде массива.

Задача 6. Написать программу для реализации комплексных чисел в алгебраической форме в виде объектов класса. Предусмотреть функции для сложения, вычитания, деления и умножения комплексных чисел.

Задача 7. Написать программу для реализации комплексных чисел в тригонометрической форме в виде объектов класса. Предусмотреть функции для сложения, вычитания, деления и умножения комплексных чисел.

Задача 8. Написать программу для реализации комплексных чисел в алгебраической и тригонометрической форме в виде объектов класса. Предусмотреть функции для перевода числа из алгебраической формы в тригонометрическую и наоборот.

Задача 9. Написать программу с функцией для вычисления косинуса от комплексного аргумента. Комплексные числа реализовать в виде объектов класса. При вычислении воспользоваться формулой $\cos(z) = \cos(x)ch(y) - i \sin(x)sh(y)$, где $z = x + iy$.

Задача 10. Написать программу с функцией для вычисления синуса от комплексного аргумента. Комплексные числа реализовать в виде объектов класса. При вычислении воспользоваться формулой $\sin(z) = \sin(x)ch(y) + i \cos(x)sh(y)$, где $z = x + iy$.

Задача 11. Написать программу для вычисления угла между двумя векторами. Векторы реализовать в виде класса.

Задача 12. Написать программу для вычисления смешанного произведения векторов (скалярное произведение вектора и векторного произведения). Векторы реализовать в виде объектов класса.

Задача 13. Написать программу для вычисления произведения матриц. Матрицы реализовать в виде объектов класса. Для вычисления произведения создать метод класса.

Задача 14. Написать программу для вычисления производной от функции вида $f(x) = \sum_{k=1}^n a_k \sin(kx)$. Коэффициенты a_k реализовать в виде поля-массива класса. Значение функции вычисляется методом класса. Производная является объектом того же класса. Для ее вычисления создать специальный метод.

Задача 15. Написать программу для вычисления производной от функции вида $f(x) = \sum_{k=0}^n a_k \cos(kx)$. Коэффициенты a_k реализовать в виде поля-массива класса. Значение функции вычисляется методом класса. Производная является объектом того же класса. Для ее вычисления создать специальный метод.

Задача 16. Написать программу для вычисления производной от функции вида $f(x) = \sum_{k=0}^n a_k \exp(-kx)$. Коэффициенты a_k реализовать в виде поля-массива класса. Значение функции вычисляется методом класса. Производная является объектом того же класса. Для ее вычисления создать специальный метод.

Задача 17. Написать программу для выполнения аппроксимации набора данных на основе регрессионной модели вида $f(x) = ax + b$. Набор данных $\{x_i\}$ и $\{y_i\}$ ($i = 1, 2, \dots, n$), на основании которых строится модель, реализовать в виде полей-массивов класса. Параметры регресси-

онной модели вычисляются как
$$a = \frac{\left(\frac{1}{n} \sum_{i=1}^n y_i\right) \left(\frac{1}{n} \sum_{i=1}^n x_i\right) - \frac{1}{n} \sum_{i=1}^n y_i x_i}{\left(\frac{1}{n} \sum_{i=1}^n x_i\right)^2 - \frac{1}{n} \sum_{i=1}^n x_i^2}$$

и
$$b = \frac{\frac{1}{n} \sum_{i=1}^n y_i x_i - a \frac{1}{n} \sum_{i=1}^n x_i^2}{\frac{1}{n} \sum_{i=1}^n x_i}$$
. Функцию $f(x) = ax + b$ реализовать в виде

метода класса.

Задача 18. Написать программу для выполнения аппроксимации набора данных на основе регрессионной модели вида $f(x) = \frac{a}{x} + b$. Набор данных $\{x_i\}$ и $\{y_i\}$ ($i = 1, 2, \dots, n$), на основании которых строится модель, реализовать в виде полей-массивов класса. Воспользоваться тем, что параметры регрессионной модели вида $f(x) = a\varphi(x) + b\psi(x)$ вычисляются как

$$a = \frac{\sum_{i=1}^n y_i \psi_i \sum_{i=1}^n \varphi_i \psi_i - \sum_{i=1}^n y_i \varphi_i \sum_{i=1}^n \psi_i^2}{\left(\sum_{i=1}^n \varphi_i \psi_i \right)^2 - \sum_{i=1}^n \varphi_i^2 \sum_{i=1}^n \psi_i^2} \text{ и } b = \frac{\sum_{i=1}^n y_i \varphi_i - a \sum_{i=1}^n \varphi_i^2}{\sum_{i=1}^n \varphi_i \psi_i}.$$

Функцию

$f(x) = \frac{a}{x} + b$ реализовать в виде метода класса.

Задача 19. Написать программу для вычисления матричного косинуса $\cos(A) = E - \frac{A^2}{2!} + \frac{A^4}{4!} - \frac{A^6}{6!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n A^{2n}}{(2n)!}$. Необходимо создать специальный класс с соответствующим методом.

Задача 20. Написать программу для вычисления матричного синуса $\sin(A) = A - \frac{A^3}{3!} + \frac{A^5}{5!} - \frac{A^7}{7!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n A^{2n+1}}{(2n+1)!}$. Необходимо создать специальный класс с соответствующим методом.

Глава 9

Конструкторы и деструкторы

Самая тяжелая работа – та, которую мы не решаемся начать.

III. Бодлер

Читатель наверняка обратил внимание, что в рассмотренных ранее примерах значительная часть кода предназначалась для инициализации полей создаваемых объектов класса. Естественным образом возникает вопрос, нельзя ли как-то упростить процедуру присваивания значения полям объектов при их создании? Отсюда вытекает и более сложная и перспективная задача по определению базового набора операций, которые необходимо выполнить при создании (и, в принципе, при уничтожении) объектов. Реализуется такая задача через механизм использования конструкторов и деструкторов.

Концепция конструкторов и деструкторов в C++ является элегантной и достаточно эффективной. Рассмотрению вопросов, связанных с созданием и использованием конструкторов и деструкторов, посвящена эта глава.

Конструктор – метод, который автоматически вызывается при создании объекта. Деструктор – метод, вызываемый автоматически при выгрузке объекта из памяти. Существуют конструкторы и деструкторы по умолчанию, однако их можно определить и в явном виде. Начнем с создания конструктора.

Создание и перегрузка конструктора

Конструктор в классе создается практически так же, как и обычные методы, с учетом двух принципиальных особенностей:

1. Имя конструктора совпадает с именем класса.
2. Конструктор не возвращает результат и для него тип результата не указывается вообще.

Во всем остальном конструктор напоминает обычный метод, за исключением той лишь разницы, что метод необходимо вызывать в явном виде, а конструктор вызывается автоматически и только при создании объекта.

Конструктор может иметь аргументы, а может и не иметь. Кроме того, конструктор можно перегружать. В листинге 9.1 приведен пример создания конструктора.

Листинг 9.1. Создание конструктора

```

#include <iostream>
using namespace std;
class MyClass{
public:
    int m,n;
    //Конструктор класса:
    MyClass() {
        m=0;
        n=0; }
    void show() {
        cout<<"m = "<<m<<endl;
        cout<<"n = "<<n<<endl; }
};

int main() {
    //При создании класса поля получают значения:
    MyClass obj;
    //Отображение полей объекта:
    obj.show();
    return 0;
}

```

У класса `MyClass` два поля `m` и `n`, метод `show()` для отображения значения полей и явно созданный конструктор. Синтаксис объявления конструктора следующий:

```

MyClass() {
    m=0;
    n=0; }

```

Как отмечалось, имя конструктора `MyClass()` совпадает с именем класса, аргументов у конструктора нет, но круглые скобки после имени метода все равно указываются. В теле конструктора полям `m` и `n` присваиваются нулевые значения. Это означает, что при создании нового объекта полям объекта присваиваются нулевые значения. Убедиться в том несложно: в методе `main()` программы командой `MyClass obj` создается объект `obj`, после чего сразу на экран командой `obj.show()` отображаются значения полей созданного объекта. В результате получим:

```

m = 0
n = 0

```

Обращаем внимание, что формально процедура создания объекта не изменилась, зато изменился результат. Это удобно, поскольку при создании объекта поля имеют значения, поэтому в случае несанкционированного доступа к полям объекта есть шанс получить разумный результат. Если касаться конкретного

рассмотренного примера, то некоторое неудобство состоит в том, что для всех объектов поля получают одни и те же значения. Хорошо было бы предусмотреть возможность в достаточно простой форме задавать явно значения полей при создании объекта. Проблема решается с помощью передачи конструктору аргументов. Пример измененного программного кода приведен в листинге 9.2.

Листинг 9.2. Конструктор с аргументами

```
#include <iostream>
using namespace std;
class MyClass{
public:
    int m,n;
    //Конструктор класса с аргументами:
    MyClass(int a,int b){
        m=a;
        n=b;}
    void show(){
        cout<<"m = "<<m<<endl;
        cout<<"n = "<<n<<endl;}
};
int main(){
    //При создании объекта указываются значения полей:
    MyClass obj(1,2);
    //Отображение значения полей объекта:
    obj.show();
    return 0;
}
```

Во-первых, в описании класса изменен код конструктора:

```
MyClass(int a,int b){
    m=a;
    n=b;}
```

Теперь у конструктора два целочисленных аргумента, которые присваиваются в качестве значений полям объекта. Во-вторых, при создании объекта в главном методе программы после имени объекта в круглых скобках указываются аргументы, которые передаются конструктору. В этом смысле команда `MyClass obj(1,2)` означает создание объекта `obj`, и при его создании вызывается конструктор с аргументами 1 и 2. Поэтому результат выполнения программы такой:

```
m = 1
n = 2
```

Здесь, правда, появляется другая проблема – поскольку конструктор определен с двумя аргументами, при создании нового объекта каждый раз при-

дется указывать аргументы для конструктора. Это тоже не очень удобно. Поэтому, как правило, в классе создают несколько вариантов конструкторов или, что то же самое, выполняют перегрузку конструктора.

Правила перегрузки конструктора такие же, как и правила перегрузки методов и функций. Следует только помнить, что тип результата для конструктора не указывается, поэтому разные варианты конструкторов могут отличаться количеством и типом аргументов. Программный код с перегруженным конструктором приведен в листинге 9.3.

Листинг 9.3. Перегрузка конструктора

```
#include <iostream>
using namespace std;
class MyClass{
public:
    int m,n;
    //Конструктор класса без аргументов:
    MyClass(){
        m=0;
        n=0;}
    //Конструктор класса с одним аргументом:
    MyClass(int a){
        m=a;
        n=a;}
    //Конструктор класса с двумя аргументами:
    MyClass(int a,int b){
        m=a;
        n=b;}
    void show(){
        cout<<"m = "<<m<<endl;
        cout<<"n = "<<n<<endl;}
};

int main(){
    //Разные способы создания объектов:
    MyClass obj1;
    MyClass obj2(1);
    MyClass obj3(2,3);
    //Отображение значения полей объектов:
    obj1.show();
    obj2.show();
    obj3.show();
    return 0;
}
```

В программе объявлено три варианта конструктора: без аргумента, с одним аргументом, с двумя аргументами. Поэтому создавать объекты можно тре-

мя разными способами: не указывая аргументы для конструктора (команда `MyClass obj1`), указав один аргумент (команда `MyClass obj2(1)`) или с двумя аргументами (команда `MyClass obj3(2, 3)`). Если аргументы конструктору не переданы, полям объекта присваиваются нулевые значения. В случае, если вызывается конструктор с одним аргументом, полям присваиваются одинаковые значения, которые определяются аргументом конструктора. При передаче двух аргументов конструктору эти аргументы определяют значения полей объекта. Результат выполнения программы имеет вид:

```
m = 0
n = 0
m = 1
n = 1
m = 2
n = 3
```

Таким образом, наличие перегруженного конструктора существенно повышает гибкость программного кода.

Остановимся еще на одном виде конструкторов: это конструктор для создания нового объекта на основе уже существующего объекта. В этом случае аргументом конструктора является объект того же класса, в котором описан конструктор. Например, для рассмотренного выше примера вариант конструктора для создания объекта на основе уже существующего может выглядеть так:

```
MyClass(MyClass obj) {
    m=obj.m;
    n=obj.n; }
```

Конструкторы незаменимы с практической точки зрения, их применение позволяет легко и быстро решать сложные задачи. К конструкторам мы еще будем возвращаться неоднократно и в разном контексте. Но прежде остановимся на том, как создаются и используются деструкторы.

Использование деструкторов

Игра не закончена, пока она не закончена.

Й. Берра

При выгрузке объекта из памяти автоматически вызывается метод, который называется деструктором. Деструкторы, как и конструкторы, можно явно описывать при создании класса. Правила создания деструкторов еще более консервативны, чем правила создания конструкторов. Они следующие:

1. Имя деструктора совпадает с именем класса, но перед именем деструктора указывается символ «тильда» ~.
2. Тип результата для деструктора не указывается (как и для конструктора).
3. У деструктора нет аргументов.

Из перечисленных правил следует, что деструктор не может быть перегружен: у перегруженных функций и методов должны совпадать названия, но отличаться прототипы. У деструктора имя определено однозначно, аргументов нет, результата нет, поэтому нет технической возможности к тому, чтобы создать в одном классе деструкторы с разными прототипами. В листинге 9.4 приведен пример использования деструктора.

Листинг 9.4. Использование деструктора

```
#include <iostream>
using namespace std;
class MyClass{
public:
    int m,n;
    //Конструктор класса:
    MyClass() {
        m=0;
        n=0;
        cout<<"Object has been created"<<endl;}
    //Деструктор класса:
    ~MyClass() {
        cout<<"Object has been deleted"<<endl;}
};
int main(){
    MyClass obj;
    return 0;
}
```

Программный код предельно прост. В классе `MyClass`, кроме двух целочисленных полей, предусмотрен конструктор без аргументов, которым присваиваются нулевые значения полям объекта и выводится сообщение о создании объекта, а также деструктор, которым выводится сообщение об удалении объекта. Главный метод программы содержит инструкцию создания объекта `MyClass obj`, после чего работа программы завершается. В результате выполнения программы появятся две строки:

```
Object has been created
Object has been deleted
```

Объясняется все просто: первая строка есть следствие создания объекта. Она появляется в результате вызова конструктора. Со второй строкой дела обстоят немного интереснее. Дело в том, что при завершении программы из памяти выгружаются все созданные в программе объекты. Выгружается из памяти и созданный объект `obj`. При выгрузке объекта из памяти вызывается деструктор, благодаря чему и появляется сообщение об удалении объекта.

Совместное использование конструкторов и деструкторов нередко приводит к неожиданным, на первый взгляд, результатам. В качестве иллюстрации рассмотрим программный код, представленный в листинге 9.5.

Листинг 9.5. Использование конструкторов и деструкторов

```
#include <iostream>
using namespace std;
class MyClass{
public:
    int m,n;
    //Конструктор класса:
    MyClass(int a,int b){
        m=a;
        n=b;
        cout<<"Object "<<this<<" has been created"<<endl;
        cout<<"m = "<<m<<endl;
        cout<<"n = "<<n<<endl;
    }
    //Деструктор класса:
    ~MyClass() {
        cout<<"Object "<<this<<" has been deleted"<<endl;
    };
    //В функции создается объект:
    MyClass MyFunc(int x,int y){
        MyClass tmp(x,y);
        return tmp;}
int main(){
    //Создание объекта:
    MyClass obj(1,2);
    //Изменение объекта:
    obj=MyFunc(3,4);
    return 0;
}
```

У класса `MyClass` есть конструктор (с передачей двух аргументов) и деструктор. Сразу обращаем внимание читателя, что в конструкторе и деструкторе, помимо текстового сообщения о создании или удалении объекта, указывается также и его адрес. Адрес объекта в обработке получаем с помо-

щью указателя `this`. Такой подход удобен, поскольку позволяет отслеживать создание и удаление различных объектов.

Также в программе описывается функция `MyFunc()`, у которой два целочисленных аргумента. В функции создается объект класса `MyClass` с аргументами конструктора, совпадающими с аргументами функции. В качестве значения функции возвращается этот объект.

В главном методе программы командой `MyClass obj(1, 2)` создается объект `obj`. Затем командой `obj=MyFunc(3, 4)` этот объект изменяется (переопределяются значения его полей). Результатом выполнения программы будет последовательность сообщений (адреса объектов могут меняться от запуска к запуску, однако в соответствующих местах адреса все равно будут совпадать):

```
Object 0012FF6C has been created
m = 1
n = 2
Object 0012FEE8 has been created
m = 3
n = 4
Object 0012FEE8 has been deleted
Object 0012FF64 has been deleted
Object 0012FF6C has been deleted
```

Поясним, почему сообщения именно такие. Для этого рассмотрим внимательнее программный код метода `main()`.

В результате выполнения команды `MyClass obj(1, 2)` появляется первый блок сообщений

```
Object 0012FF6C has been created
m = 1
n = 2
```

Здесь все достаточно просто: при создании объекта вызывается конструктор, следствием выполнения которого и является означенный блок сообщений. Чтобы понять происхождение второй части результата, необходимо учесть особенности вызова функции `MyFunc()`. Эта функция вызывается при выполнении команды `obj=MyFunc(3, 4)`. Выполнение команды подразумевает вычисление выражения `MyFunc(3, 4)`. Вызывается функция `MyFunc()`, которой аргументами передаются числа 3 и 4. Функцией создается временный объект `tmp`, следствием чего является блок сообщений

```
Object 0012FEE8 has been created
m = 3
n = 4
```

Однако это не единственный объект, который создается функцией. Дело в том, что механизм возвращения функцией результата подразумевает создание временного объекта, в который записывается результат. Поэтому в программе мы имеем дело с тремя объектами: один явно создается в главном методе программы, один локальный объект явно создается в функции, и один объект создается функцией неявно для записи результата функции. При удалении каждого из этих объектов вызывается деструктор, в силу чего появляются соответствующие сообщения. Возникает естественный вопрос: почему конструктор вызывается два раза, а деструктор – три? В частности, сообщения о создании объекта с адресом 0012FF64 нет, зато есть сообщение о его удалении. Для ответа на этот вопрос необходимо отметить, что деструктор вызывается при удалении объекта из памяти, а конструктор вызывается при объявлении объекта. Объявлений объектов в программе два, поэтому конструктор вызывается дважды. Деструктор вызывается три раза: первый раз при удалении из памяти локального объекта в функции, второй раз при удалении временного объекта с результатом функции (как только функция возвращает результат, объект результата удаляется) и третий раз при завершении работы программы, когда освобождается место от объекта, созданного в главном методе программы.

Вызов конструктора

Ранее неоднократно отмечалось, что конструктор класса вызывается автоматически при объявлении объекта класса. Это вовсе не означает, что после создания объекта нельзя еще раз вызвать конструктор. Для понимания механизмов возможного использования конструктора (кроме автоматического его вызова при объявлении объекта) необходимо принять к сведению, что вызов конструктора подразумевает создание объекта соответствующего класса. Если конструктор вызывается при объявлении объекта, то созданный в результате вызова конструктора объект фактически является объявляемым объектом. Несколько иная ситуация при вызове конструктора после создания объекта. Речь идет о команде вида `объект=конструктор(аргументы)`. В этом случае инструкцией `конструктор(аргументы)` в соответствии с переданными конструктору аргументами создается новый безымянный объект, а потом этот объект присваивается в качестве значения объекту, указанному слева от оператора присваивания. Напомним, что при присваивании объектов выполняется побитовое копирование. После копирования безымянный объект удаляется из памяти. Ситуацию иллюстрирует листинг 9.6.

Листинг 9.6. Явный вызов конструктора

```

#include <iostream>
using namespace std;
//Класс с закрытым полем, конструктором и деструктором:
class Demo{
    int n;
public:
    Demo(int m){
        n=m;
        cout<<"Constructor: "<<this<<endl;}
    void setn(int m){n=m;}
    void getn(){
        cout<<"n = "<<n<<endl;
        cout<<"Object: "<<this<<endl;
    }
    ~Demo(){
        cout<<"Destructor: "<<this<<endl;}
};

int main(){
    //Создание объекта:
    Demo a(1);
    //Создание объекта с явным вызовом конструктора:
    Demo b=Demo(2);
    //Проверка значений:
    a.getn();
    b.getn();
    //Изменение значения поля с помощью метода класса:
    a.setn(-1);
    //Проверка значений:
    a.getn();
    //Копирование объектов:
    a=b;
    //Проверка результата копирования:
    a.getn();
    //Изменение значения поля с помощью конструктора:
    a=Demo(0);
    //Проверка результата:
    a.getn();
    return 0;
}

```

Результат выполнения программы может иметь следующий вид (жирным шрифтом выделены адреса объектов, которые от запуска к запуску могут изменяться):

```

Constructor: 0012FF70
Constructor: 0012FF6C
n = 1
Object: 0012FF70
n = 2
Object: 0012FF6C
n = -1
Object: 0012FF70
n = 2
Object: 0012FF70
Constructor: 0012FF68
Destructor: 0012FF68
n = 0
Object: 0012FF70
Destructor: 0012FF6C
Destructor: 0012FF70

```

Проанализируем полученные данные. Для этого обратимся к главному методу программы. В методе `main()` вначале объявляются два объекта класса `Demo`: командой `Demo a(1)` создается объект `a` со значением поля `n`, равном 1, и командой `Demo b=Demo(2)` объект `b` со значением поля `n`, равном 2. Во втором случае использован явный вызов конструктора. Конструктор при объявлении объекта вызывается в любом случае. Для класса `Demo` конструктор прописан явно и кроме присваивания значения закрытому полю `n` конструктором выводится сообщение `Constructor...` с адресом созданного объекта. При создании двух указанных объектов появляются сообщения `Constructor: 0012FF70` и `Constructor: 0012FF6C` соответственно. Далее из объектов `a` и `b` вызывается метод `getn()`, которым отображается значение поля `n` объекта и его адрес. Этому фрагменту кода соответствует блок

```

n = 1
Object: 0012FF70
n = 2
Object: 0012FF6C

```

Командой `a.setn(-1)` изменяется значение поля `n` объекта `a`, после чего выполняется проверка результата командой `a.getn()`. В результате получаем

```

n = -1
Object: 0012FF70

```

Здесь стоит обратить внимание, что адрес объекта `a` не изменился – что вполне ожидаемо. Не меняется адрес объекта `a` и при копировании объектов (команды `a=b` и `a.getn()`, которым соответствует результат `n = 2` и `Object: 0012FF70`). Блок результата `Constructor: 0012FF68` и `Destructor: 0012FF68` появляется благодаря команде `a=Demo(0)`. В результате выполнения этой команды создается новый безымянный объ-

ект (его адрес в данном случае 0012FF68) с нулевым значением поля `n`, этот объект копируется в объект `a`, после чего удаляется из памяти. Выполнив команду `a.getn()` убеждаемся, что поле `n` объекта `a` получило значение 0, а адрес объекта не изменился. При завершении работы программы вызываются деструкторы для удаления объектов `b` и `a`.

Конструктор создания копии

– Гигантский номер! Посмотрите на эту маску – это же точная копия Огурцова!

– Какая копия? Я и есть Огурцов!

Из к/ф «Карнавальная ночь»

При инициализации объекта в формате `класс объект1=объект2` для объекта `объект1` выделяется место и затем выполняется побитовое копирование объекта `объект2` в объект `объект1`. Во многих случаях это удобно, но не всегда. В C++ существует возможность переопределять конструктор для создания копии объекта при инициализации. К такому переопределению прибегают в силу разных причин: часто чтобы избежать возможных проблем в процессе выполнения программы, а иногда и для того, чтобы упростить структуру программы. Именно такой случай рассмотрим далее. В листинге 9.7 приведен программный код, содержащий, кроме прочего, переопределение конструктора создания копии объекта.

Листинг 9.7. Конструктор создания копии

```
#include <iostream>
using namespace std;
class Demo{
public:
    int n;
    int m;
    Demo(int i,int j){
        n=i;
        m=j;}
    Demo(Demo &obj){
        n=2*obj.n;
        m=obj.m+3;}
    void show(){
        cout<<"n = "<<n<<endl;
        cout<<"m = "<<m<<endl;}
};
int main(){
```

```

int i;
Demo a(1, 0);
Demo b=a;
a.show();
b.show();
for(i=1; i<=4; i++) {
a=Demo(a); }
a.show();
return 0;
}

```

Прежде чем приступить к анализу программного кода, кратко остановимся на особенностях конструкторов создания копий. Конструктор создания копии – это конструктор, аргументом которого является объект того же класса. Объект-аргумент конструктора передается только по ссылке. Причина кроется в том, что конструктор создания копии вызывается при инициализации объекта. Напомним, что в этом случае конструктором создается безымянный объект, который затем интерпретируется как создаваемый (инициализируемый при объявлении). Если аргумент функции (в том числе и конструктору) передается по значению, то на самом деле для объекта-аргумента создается копия. Предположим, в конструкторе создания копии аргумент передается по значению. В этом случае для создания копии аргумента вызывается конструктор создания копии. У этого конструктора создания копии есть аргумент, для которого необходимо было бы создавать копию. Создание копии подразумевает вызов все того же конструктора, и так далее. Передача аргумента по ссылке предотвращает такую рекурсивную неприятность.

Используемый по умолчанию конструктор создания копии, выполняющий побитовое копирование при создании объекта, не всегда приемлем. Классический пример такой ситуации – создание копии объекта, полем которого является указатель на третий объект. В этом случае два объекта через значения своих полей ссылаются на один и тот же объект, что может привести к серьезным проблемам, особенно если третий объект создавался динамически. В таких и подобных им ситуациях выполняется переопределение конструктора создания копии.

Хотя конструктор формально и называется конструктором создания копии, его можно переопределить так, что на основе объекта, переданного конструктору в качестве аргумента, будет создаваться совершенно иной по значениям своих полей объект (именно такой подход использован в коде в листинге 9.7).

Что касается рассматриваемого примера, то у класса `Demo` два целочисленных поля `n` и `m`, а также метод `show()` для отображения значений полей.

Кроме этих полей и метода, в классе описаны два варианта конструкторов: с двумя аргументами типа `int` и с одним аргументом типа `Demo` (конструктор создания копии). Если с первым вариантом конструктора все просто и понятно, то конструктор создания копии требует некоторых пояснений. Командой `n=2*obj.n` (`obj` – аргумент конструктора) полю `n` создаваемого объекта присваивается удвоенное значение соответствующего поля объекта-аргумента. Поле `m` создаваемого объекта определяется командой `m=obj.m+3` (к значению поля `m` объекта-аргумента присваивается значение 3). Теперь, обратившись к коду главного метода программы `main()`, несложно предугадать результат выполнения программы:

```
n = 1
m = 0
n = 2
m = 3
n = 16
m = 12
```

В программе сначала командой `Demo a(1,0)` создается объект `a` класса `Demo` со значениями полей `n=1` и `m=0`. На основе этого объекта командой `Demo b=a` создается объект `b`. Поскольку при создании объекта `b` используется конструктор создания копии, значения полей этого объекта `n=2` и `m=3`. Отметим, что команда `Demo b=a` благодаря автоматическому приведению типов эквивалентна команде `Demo b=Demo(a)`. После проверки значений полей объектов `a` и `b` (команды `a.show()` и `b.show()`) четыре раза подряд выполняется команда `a=Demo(a)`. В этом случае явно вызывается конструктор создания копии. Каждый раз при его вызове поле `n` умножается на два, а к полю `m` прибавляется число 3. В результате значения полей объекта `a` равны `n=16` и `m=12`.

В заключение заметим, что аналогичным образом можно переопределить и операцию присваивания объектов, однако реализуется это путем переопределения оператора присваивания. Переопределению операторов посвящена следующая глава.

Примеры решения задач

Я не могу думать, когда я сосредотачиваюсь.

Й. Берра

В приведенных далее примерах представлены программы, в которых используются конструкторы и деструкторы.

■ Создание бинарного дерева

Конструкторы удобно использовать для создания различных связанных структур, в том числе и бинарных деревьев. Напомним, что бинарным деревом называется система объектов, связанных по следующему принципу. В вершине иерархии находится один родительский объект, и каждый объект структуры ссылается на два объекта того же типа. В листинге 9.8 приведен пример программного кода с определением класса `BinTree`. Класс содержит конструктор, через который и реализуется процесс создания бинарного дерева. Конструктор вызывается с аргументом целого типа. При этом аргумент конструктора определяет «глубину» (т.е. количество уровней) бинарного дерева.

Листинг 9.8. Создание бинарного дерева

```
#include <iostream>
using namespace std;
class BinTree{
public:
    static int Count;
    int m;
    BinTree *p1;
    BinTree *p2;
    BinTree(int n){
        if(n==1){
            p1=NULL;
            p2=NULL;}
        else{
            p1=new BinTree(n-1);
            p2=new BinTree(n-1);}
        m=++Count;
        cout<<"Object created: "<<this<<" : "<<m;
        cout<<" -> Number of objects: "<<Count<<endl;}
    ~BinTree(){
        delete p1;
        delete p2;
        Count--;
        cout<<"Object deleted: "<<this<<" : "<<m;
        cout<<" -> Number of objects: "<<Count<<endl;}
};

int BinTree::Count;
int main(){
    BinTree::Count=0;
    BinTree obj1(3);
    BinTree *p;
```

```

p=new BinTree(2);
delete p;
return 0;}

```

При создании и удалении объектов выполняется их автоматический подсчет. Реализуется это через статическое целочисленное поле `Count` класса `BinTree`. Кроме этого, в классе объявляется целочисленное поле `m` для нумерации объектов. Класс также содержит в качестве полей два указателя `p1` и `p2` на объекты класса `BinTree`. Поля используются для связи объекта с двумя другими объектами того же класса. Через эти поля-указатели реализуется вся структура бинарного дерева. Вся функциональность класса `BinTree` реализуется через конструктор и деструктор.

Конструктору передается целочисленный аргумент. Если аргумент равен единице, полям `p1` и `p2` создаваемого объекта присваиваются значения `NULL` (нулевые указатели). В противном случае динамически создается два объекта класса `BinTree`, при этом аргумент для конструктора создаваемых объектов уменьшается на единицу.

При создании нового объекта класса `BinTree` значение статической переменной `Count` увеличивается на единицу, и это значение присваивается полю `m`. Соответствующая команда в конструкторе имеет вид `m=++Count`. Также создание объекта сопровождается выводом соответствующего сообщения с указанием адреса выделенной под объект памяти, значения поля `m` и общего количества созданных объектов.

В деструкторе освобождается место памяти, выделенное для объектов, на которые указывают поля `p1` и `p2` удаляемого объекта, уменьшается на единицу значение статического поля `Count`, а также выводится сообщение с указанием адреса удаляемого объекта, значения его поля `m`, равно как и количество объектов, оставшихся в работе.

В главном методе программы статическое поле `Count` инициализируется с начальным нулевым значением, а также создается два объекта (точнее, два бинарных дерева) класса `BinTree` – один статически, один динамически. Результат выполнения программы может выглядеть так, как показано ниже:

```

Object created: 003210C8 : 1 -> Number of objects: 1
Object created: 00321180 : 2 -> Number of objects: 2
Object created: 00321090 : 3 -> Number of objects: 3
Object created: 003211F0 : 4 -> Number of objects: 4
Object created: 00322850 : 5 -> Number of objects: 5
Object created: 003211B8 : 6 -> Number of objects: 6
Object created: 0012FF68 : 7 -> Number of objects: 7
Object created: 003228C0 : 8 -> Number of objects: 8
Object created: 003228F8 : 9 -> Number of objects: 9

```

```

Object created: 00322888 : 10 -> Number of objects: 10
Object deleted: 003228C0 : 8 -> Number of objects: 9
Object deleted: 003228F8 : 9 -> Number of objects: 8
Object deleted: 00322888 : 10 -> Number of objects: 7
Object deleted: 003210C8 : 1 -> Number of objects: 6
Object deleted: 00321180 : 2 -> Number of objects: 5
Object deleted: 00321090 : 3 -> Number of objects: 4
Object deleted: 003211F0 : 4 -> Number of objects: 3
Object deleted: 00322850 : 5 -> Number of objects: 2
Object deleted: 003211B8 : 6 -> Number of objects: 1
Object deleted: 0012FF68 : 7 -> Number of objects: 0

```

Сначала создаются десять объектов (семь в результате выполнения команды `BinTree obj1(3)` – дерево из трех уровней содержит $2^3 - 1 = 7$ элементов, и три в результате динамического создания объекта командой `p=new BinTree(2)` – создается $2^2 - 1 = 3$ объектов). Затем динамический объект удаляется, в результате чего удаляются все три объекта (с номерами 10, 9 и 8), входящие в соответствующее бинарное дерево. Статический объект со всей связанной с ним структурой (объекты со значениями поля `m` от 1 до 7) удаляется при завершении работы программы автоматически.

Обращаем внимание, что если создается несколько деревьев, нумерация в них используется общая. Например, первый элемент второго дерева в рассмотренном примере получает номер 8.

■ Ряд для экспоненты

В следующем примере реализуется вычисление экспоненты, правда, несколько необычным образом. Для этих целей создается класс, у которого имеется поле – динамический массив. Элементами массива являются слагаемые ряда, через который вычисляется экспонента. Массив заполняется при создании соответствующего объекта. Аргументами конструктору передаются верхний индекс ряда `int n` и аргумент экспоненты `double x`. Программный код приведен в листинге 9.9.

Листинг 9.9. Ряд для экспоненты

```

#include <iostream>
using namespace std;
class MyExp{
public:
    int n;
    double *p;
    MyExp(int i, double x){
        n=i;

```

```

    p=new double[n+1];
    p[0]=1;
    for(int k=1;k<=n;k++) p[k]=p[k-1]*x/k;
    }
    ~MyExp(){delete [] p;}
};

int main(){
    int n,i;
    double x,s=0;
    cout<<"enter n= ";
    cin>>n;
    cout<<"enter x= ";
    cin>>x;
    MyExp obj(n,x);
    for(i=0;i<=n;i++) s+=obj.p[i];
    cout<<"exp("<<x<<" )= "<<s<<endl;
    return 0;}

```

Результат выполнения программы может иметь следующий вид (жирным шрифтом выделен ввод пользователя):

```

enter n= 100
enter x= 1
exp(1)= 2.71828

```

Пользователем вводятся значения для верхней границы ряда и аргумент экспоненты (переменные `int n` и `double x` в главном методе программы). Эти аргументы передаются конструктору при создании объекта командой `MyExp obj(n,x)`. Конструктором первый аргумент присваивается в качестве значения полю `n` объекта `obj`. Это же значение используется при выделении памяти под динамический массив, указатель на первый элемент которого присваивается в качестве значения полю `p` объекта `obj`. После выделения места под массив выполняется заполнение элементов массива. В качестве значения элементу с нулевым индексом присваивается единица, а прочие элементы заполняются на основе предыдущего значения: предыдущий элемент умножается на `x` и делится на индексную переменную `k`.

В деструкторе выполняется очистка памяти, выделенной под динамический массив.

■ Поле-объект

Полями класса могут быть не только переменные базовых типов, но и объекты. Подобная ситуация иллюстрируется в следующем примере. В листинге 9.10 представлен программный код, в котором определяется класс

с конструктором и деструктором. Впоследствии объект этого класса используется в качестве поля другого класса, также имеющего конструктор и деструктор. Пример иллюстрирует способы вызова конструкторов и деструкторов в такой нетривиальной ситуации.

Листинг 9.10. Поле-объект

```
#include <iostream>
using namespace std;
// "Внутренний" класс:
class Inner{
public:
    int n;
    // Конструктор:
    Inner(){
        n=0;
        cout<<"Inner-object created with n="<<n<<endl;}
    // Деструктор:
    ~Inner(){
        cout<<"Inner-object destroyed with n="<<n<<endl;}
};
// "Внешний" класс:
class Outer{
public:
    int m;
    // Поле-объект:
    Inner obj;
    // Перегруженный конструктор:
    Outer(int i){
        m=i;
        cout<<"Outer-object created with m="<<m<<endl;}
    Outer(int i,int j){
        m=i;
        obj.n=j;
        cout<<"Outer-object created with m="<<m<<endl;}
    // Деструктор:
    ~Outer(){
        cout<<"Outer-object destroyed with m="<<m<<endl;}
};
int main(){
    int i=10,j=20,k=30;
    Outer a(i);
    Outer b(j,k);
    cout<<"a.m: "<<a.m<<" a.obj.n: "<<a.obj.n<<endl;
    cout<<"b.m: "<<b.m<<" b.obj.n: "<<b.obj.n<<endl;
    return 0;}
```


В программе объявляется класс `Inner`, имеющий целочисленное поле `n`, конструктор (без аргументов) и деструктор. При создании и уничтожении объекта выводится соответствующее сообщение с указанием значения поля `n` создаваемого или уничтожаемого объекта.

В классе `Outer` описано целочисленное поле `m`, а также поле `obj`, являющееся объектом класса `Inner`. У класса `Outer` также есть перегруженный конструктор (с одним аргументом и с двумя), равно как и деструктор.

В главном методе программы создается два объекта класса `Outer`. Объект `a` создается посредством конструктора с одним аргументом, а объект `b` – с помощью конструктора с двумя аргументами. Значения полей объектов выводятся на экран. Результат выполнения программы имеет вид:

```
Inner-object created with n=0
Outer-object created with m=10
Inner-object created with n=0
Outer-object created with m=20
a.m: 10 a.obj.n: 0
b.m: 20 b.obj.n: 30
Outer-object destroyed with m=20
Inner-object destroyed with n=30
Outer-object destroyed with m=10
Inner-object destroyed with n=0
```

Обращаем внимание, что при создании объекта класса `Outer`, кроме сообщения о создании этого объекта, предварительно выводится сообщение о создании объекта `Inner` – это создается поле `obj` – объект класса `Inner`, причем создается он с помощью конструктора «по умолчанию» (конструктор без аргументов) класса `Inner`, поэтому поле `n` этого объекта получает нулевое значение. Кроме того, специфично выглядит обращение к полю `n` объекта `obj`, являющегося полем объекта класса `Outer`: например, `a.obj.n` или `b.obj.n`.

■ Поле-массив объектов

Несколько более сложная ситуация имеет место, когда полем класса является массив объектов другого класса. Такой случай рассматривается в программном коде листинга 9.11.

Листинг 9.11. Поле-массив объектов

```
#include <iostream>
using namespace std;
//Размер поля-массива:
const int n=10;
```

```

// "Внутренний" класс:
class Inner{
public:
    int m;
    int number;
    // Метод для отображения полей:
    void show(){
        if(m!=1) (this-1)->show();
        cout<<number<<" ";
    };
};

// "Внешний" класс:
class Outer{
public:
    // Поле-массив объектов:
    Inner obj[n];
    // Конструктор:
    Outer(){
        int i;
        obj[0].m=1;
        obj[0].number=1;
        obj[1].m=2;
        obj[1].number=1;
        for(i=2; i<n; i++){
            obj[i].m=i+1;
            obj[i].number=obj[i-1].number+obj[i-2].number;
        }
    };
};

int main(){
    Outer a;
    a.obj[n/2].show();
    cout<<endl;
    a.obj[n-1].show();
    cout<<endl;
    return 0;
}

```

В программе реализуется последовательность чисел Фибоначчи – правда, довольно экзотическим способом. В программе описывается класс `Inner`, у которого два целочисленных поля `m` и `number`. Последовательность чисел Фибоначчи реализуется посредством объектов класса `Inner` – в поле `m` заносится порядковый номер числа, в поле `number` записывается непосредственно число, а сами объекты организуются в массив, который является полем класса `Outer`.

В классе `Outer` объявлен в качестве поля массив объектов класса `Inner`. Размер массива ранее определен в виде целочисленной константы. В конструкторе класса `Outer` выполняется заполнение полей объектов класса

`Inner` – элементов массива `obj`. Поля `m` формируют последовательность натуральных чисел, а поля `number` заполняются так: для первых двух полей значение равно 1, а поле каждого следующего объекта равно сумме полей двух предыдущих объектов.

Но самое ужасное происходит в методе `show()` класса `Inner`. Методом выводится значение поля `number` объекта, но предварительно, если поле `m` объекта, из которого вызывается метод, не равняется единице, вызывается метод `show()` «предыдущего» объекта. Для определения «предыдущего» объекта используется адресная арифметика в виде инструкции `this-1`. Эта ссылка означает объект, размещенный в предыдущей ячейке по отношению к текущему объекту. Такая команда имеет корректный смысл, только если речь идет о массиве объектов – в противном случае результат будет весьма трагическим. Таким образом, получаем своеобразную рекурсию, а действие метода `show()` сводится к тому, что на экран выводятся значения полей `number` объектов массива, от начального до текущего, из которого вызывается метод `show()`.

Результат выполнения программы имеет вид:

```
1 1 2 3 5 8
1 1 2 3 5 8 13 21 34 55
```

В первой строке командой `a.obj[n/2].show()` выводятся 6 чисел последовательности Фибоначчи (в массиве 10 элементов, метод вызывается из объекта с индексом 5, что соответствует 6-ти числам в последовательности). Второй командой `a.obj[n-1].show()` выводятся значения из всего массива.

■ Вызов в конструкторе метода

Классы используются в основном для того, чтобы скрыть от пользователя доступ к членам класса – именно поэтому по умолчанию все члены класса являются закрытыми. В качестве иллюстрации рассмотрим простой пример.

В очередной раз рассмотрим процесс вычисления экспоненты. Для записи экспоненты и ее аргумента используем поля класса. Проблема состоит в том, что при таком подходе поля должны меняться синхронно – если меняется поле-аргумент, должно соответствующим образом измениться и поле для значения экспоненты, причем второе поле изменяется только вследствие изменения первого поля. Проблема решается, если сделать поля закрытыми, а доступ к ним реализовать так, чтобы выполнялись необходимые условия. Рассмотрим листинг 9.12.

Листинг 9.12. Вызов конструктора в методе

```

#include <iostream>
using namespace std;
//Класс с закрытыми полями:
class MyExp{
    double x;
    double хexp;
    //Закрытый метод для вычисления экспоненты:
    double mexp(double z){
        double s=1,q=z;
        for(int i=1;i<=100;i++){
            s+=q;
            q*=z/(i+1);}
        return s;}
public:
    //Открытый метод для изменения значения полей:
    void set(double z){
        x=z;
        хexp=mexp(x);}
    //Конструктор с вызовом открытого метода:
    MyExp(double z){
        set(z);}
    //Перегруженный метод для отображения полей:
    double get(){
        return x;}
    double get(int k){
        return хexp;}
};

int main(){
    double x=1;
    MyExp obj(x);
    cout<<"x = "<<obj.get()<<endl;
    cout<<"exp(x) = "<<obj.get(0)<<endl;
    return 0;}

```

Открытый метод `set()` имеет один аргумент, который присваивается в качестве значения первому закрытому полю (аргумент экспоненты). На основе этого аргумента вычисляется экспонента – значение присваивается второму полю. Причем для вычисления экспоненты вызывается закрытый метод `mexp()`, в котором реализован процесс вычисления ряда для экспоненты. Таким образом, вне класса существует возможность только синхронного изменения полей класса. Для считывания значений предназначен перегруженный метод `get()`. Если методу аргумент не передается, методом возвращается значение аргумента экспоненты. Если методу передан целочисленный аргумент (не важно, какой), методом возвращается значение экспоненты.

В главном методе программы проверяется функциональность класса. Результат выполнения программы имеет вид:

```
x = 1
exp(x) = 2.71828
```

После создания объекта класса `MyExp` поля объекта могут изменяться с помощью метода `set()`. Отметим, что этот же метод задает значения полей объекта при его создании.

Резюме

1. Конструктор – метод, который автоматически вызывается при создании объекта.
2. Деструктор – метод, вызываемый автоматически при выгрузке объекта из памяти.
3. Конструктор в классе создается практически так же, как и обычные методы, с учетом двух принципиальных особенностей: имя конструктора совпадает с именем класса, конструктор не возвращает результат и для него тип результата не указывается вообще. Во всем остальном конструктор напоминает обычный метод, за исключением той лишь разницы, что метод необходимо вызывать в явном виде, а конструктор вызывается автоматически и только при создании объекта. Конструктор может иметь аргументы, а может и не иметь.
4. Правила перегрузки конструктора такие же, как и правила перегрузки методов и функций. Но поскольку тип результата для конструктора не указывается, разные варианты конструкторов могут отличаться количеством и типом аргументов.
5. Правила создания деструкторов следующие: имя деструктора совпадает с именем класса, но перед именем деструктора указывается символ «тильда» `~`, тип результата для деструктора не указывается (как и для конструктора), у деструктора нет аргументов. Деструктор не может быть перегружен.
6. Конструктор создания копии – это конструктор, аргументом которого является объект того же класса. Объект-аргумент конструктора передается только по ссылке, поскольку конструктор создания копии вызывается при инициализации объекта.

Контрольные вопросы

Мы легко забываем наши ошибки, если они никому, кроме нас, неизвестны.

Ф. Ларошфуко

1. Что такое конструктор? Когда он вызывается?
2. Что такое деструктор? Как и когда он вызывается?
3. Как создаются конструкторы?
4. Как перегружаются конструкторы?
5. Как создаются деструкторы?
6. Что такое конструктор копии, как он создается и в чем его особенности?

Задачи для самостоятельного решения

Далее предлагаются задачи для самостоятельного решения. Многие из них уже рассматривались. В данном случае при их решении необходимо воспользоваться концепцией конструкторов и деструкторов.

Задача 1. Написать программу для создания, на основе конструктора, дерева объектов, каждый из которых ссылается на два других объекта. Ссылка осуществляется через поля-указатели. Объекты организуются по следующему принципу. Начальный (первый) объект ссылается на два объекта (второй и третий), каждый из которых ссылается на четвертый общий объект и друг на друга. Четвертый объект ссылается на два объекта и т.д.

Задача 2. Написать программу для создания, на основе конструктора, дерева объектов, каждый из которых ссылается на два других объекта. Ссылка осуществляется через поля-указатели. Начальный (первый) объект ссылается на два объекта (второй и третий), каждый из которых ссылается друг на друга и на еще один объект (четвертый и пятый). Четвертый и пятый объекты ссылаются друг на друга и на один общий объект (шестой), который ссылается на два объекта, и т.д.

Задача 3. Написать программу для создания, на основе конструктора, дерева из объектов двух типов. Объекты первого типа содержат поля-

указатели для ссылок на один объект, а объекты второго типа имеют поля-указатели на два объекта. Начальными в конструкции являются два объекта первого типа (первый и второй). Каждый из них ссылается на объект второго типа (третий и четвертый). Эти объекты ссылаются друг на друга и на объекты первого типа (пятый и шестой) и т.д.

Задача 4. Написать программу для создания, на основе конструктора, дерева из объектов двух типов. Объекты первого типа имеют поля-указатели для ссылки на два объекта, а объекты второго типа могут ссылаться только на один объект. Начальный (первый) объект первого типа ссылается на два объекта второго типа (второй и третий), каждый из которых ссылается на четвертый общий объект первого типа. Четвертый объект ссылается на два объекта второго типа и т.д.

Задача 5. Написать программу для создания, на основе конструктора, триарного дерева объектов. В такой структуре каждый объект ссылается на три объекта такого же типа. Каждый из этих объектов, в свою очередь, ссылается на три объекта и т.д.

Задача 6. Написать программу для создания, на основе конструктора, N-кратного дерева объектов. В такой структуре каждый объект ссылается на N объектов такого же типа. Каждый из этих объектов, в свою очередь, ссылается на N объектов и т.д. Параметр N задается как константа. Поля-указатели объекта реализовать в виде массива.

Задача 7. Написать программу для создания, на основе конструктора, дерева из объектов двух типов. Объекты первого типа могут ссылаться на два объекта. Объекты второго типа могут ссылаться только на один объект. В вершине иерархии находится объект первого типа. Он ссылается на два объекта: один объект первого типа и один объект второго типа. Объект первого типа ссылается на два объекта: первого и второго типа. Объект второго типа ссылается на объект второго типа.

Задача 8. Написать программу для создания, на основе конструктора, дерева из объектов двух типов. Объекты первого типа могут ссылаться на два объекта, а объекты второго типа могут ссылаться только на один объект. Ссылки осуществляются через поля-указатели. Объекты организуются по следующему принципу. Начальный (первый) объект относится к первому типу, и он ссылается на два объекта второго типа (второй и третий объекты), каждый из которых еще на один объект второго типа (четвертый и пятый). Четвертый и пятый объекты ссылаются на один общий объект первого типа (шестой), который ссылается на два объекта второго типа, и т.д.

Задача 9. Написать программу для создания, на основе конструктора, дерева из объектов двух типов. Объекты первого типа ссылаются на два объекта. Объекты второго типа не содержат полей для выполнения ссылок. Объекты первого типа образуют последовательный список. Каждый из объектов этого списка ссылается на следующий объект и на объект второго типа.

Задача 10. Написать программу для создания, на основе конструктора, дерева из объектов двух типов. Объекты первого типа ссылаются на два объекта. Объекты второго типа ссылаются на один объект. Объекты первого и второго типов образуют два последовательных списка. Каждый из объектов этих списков ссылается на следующий объект. Кроме того, объекты первого списка (составленного из объектов первого типа) ссылаются на соответствующие объекты второго списка.

Задача 11. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления косинуса

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}.$$

Задача 12. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления синуса

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}.$$

Задача 13. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления логарифма

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \dots = \sum_{n=1}^{\infty} \frac{(-1)^{n+1} x^n}{n}.$$

Задача 14. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления гиперболического косинуса

$$ch(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}.$$

Задача 15. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления гиперболического синуса

$$sh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}.$$

Задача 16. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления ряда

$$1 + 2x + 3x^2 + 4x^3 + \dots = \sum_{n=0}^{\infty} (n+1)x^n = \frac{1}{(1-x)^2}.$$

Задача 17. По аналогии с примером «Ряд для экспоненты», написать программу для вычисления ряда

$$1 - 2x + 3x^2 - 4x^3 + \dots = \sum_{n=0}^{\infty} (-1)^n (n+1)x^n = \frac{1}{(1+x)^2}.$$

Задача 18. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления ряда

$$\frac{\sin(x)}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n+1)!}.$$

Задача 19. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления ряда

$$1 + 2^2x + 3^2x^2 + 4^2x^3 + \dots = \sum_{n=1}^{\infty} n^2 x^{n-1} = \frac{1+x}{(1-x)^3}.$$

Задача 20. По аналогии с примером «Ряд для экспоненты» написать программу для вычисления ряда

$$1 + 3x^2 + \frac{5x^4}{2!} + \frac{7x^6}{3!} + \dots = \sum_{n=0}^{\infty} \frac{(2n+1)x^{2n}}{n!} = (1+2x^2)\exp(x^2).$$

Глава 10

Перегрузка операторов

*Подъехав к развилке на дороге,
следуйте по ней.*

Йоги Берра

Переопределение операторов – одна из наиболее интересных тем, поскольку в ней в полной мере проявляется красота и гибкость языка программирования C++. Идею, положенную в основу концепции переопределения операторов, проиллюстрируем на простом примере. Допустим, что для работы с комплексными числами создается специальный класс (на самом деле в C++ для работы с комплексными числами существует класс `complex`, однако в данном случае это не принципиально).

Спецификация класса такова, что он содержит два поля: действительную и мнимую части комплексного числа. При сложении двух комплексных чисел, как известно, получаем комплексное число, действительная часть которого равна сумме действительных частей исходных чисел, а мнимая часть равна соответственно сумме мнимых частей. Безусловно, для сложения комплексных чисел, реализованных через объекты созданного пользователем класса, можно было бы создать специальную функцию, аргументами которой указываются комплексные числа (объекты), сумма которых вычисляется. Однако это не очень удобно в силу нескольких причин. Главная из них состоит в том, что интерфейс вызова операции сложения комплексных чисел в этом случае существенно отличается от сложения действительных чисел, которое реализуется с помощью оператора сложения `+`. К счастью, через механизм переопределения операторов существует возможность доопределить операцию сложения комплексных чисел (с точки зрения языка C++ это означает сложение объектов, которые принадлежат соответствующему классу) так, что можно будет для этого, как и при сложении обычных (некомплексных) чисел, использовать оператор сложения `+`. Более обще, механизм переопределения операторов позволяет использовать эти операторы для выполнения операций над объектами, которые принадлежат к классам, созданным пользователями. При работе со встроенными типами данных действие переопределенных операторов изменений не претерпевает.

Технически переопределение операторов осуществляется путем создания операторных функций. Главное отличие операторных функций от обычных, которые рассматривались ранее, состоит в том, что **формально называнием операторной функции является переопределяемый оператор**.

Эта особенность прослеживается в синтаксисе определения операторной функции. Однако важное значение имеет также то, принадлежит ли операторная функция определенному классу (тому, для работы с объектами которого переопределяется соответствующий оператор) или является внешней. Данный факт определяет в рамках процедуры переопределения оператора способ ссылки на его операнды (для бинарных операторов) или операнд (для унарных операторов). Сначала рассмотрим ситуацию, когда операторы переопределяются внешними функциями.

Внешняя операторная функция для переопределения бинарного оператора

*Пытался часто я лукавить и хитрить,
Но каждый раз судьба мой посрамляла опыт.*

Омар Хайям

При переопределении оператора внешней функцией имеет значение доступность тех полей (и методов) класса, к которым обращается создаваемая операторная функция. Если среди членов класса, к которым обращается операторная функция, есть закрытые, операторная функция должна быть дружественной функцией класса. Сейчас остановимся на случае, когда оператор переопределяется для открытых полей внешней функцией. Синтаксис объявления операторной функции при этом следующий:

```
Тип_результата operator оператор (аргументы: тип и название)
{
    программный код
}
```

Сначала указывается тип результата (Тип_результата), который возвращается операторной функцией – как и в случае объявления обычной функции. Ключевое слово `operator` является обязательным признаком операторной функции. После этого слова указывается непосредственно тот оператор, который переопределяется операторной функцией (оператор). Это может быть, например, оператор `+`, `-`, `*` или `/`, хотя этим набор допустимых для переопределения операторов не ограничивается. После оператора в скобках указывают аргументы операторной функции.

Аргументами операторной функции являются операнды. Поэтому при переопределении бинарных операторов операторной функции (внешней) передают два аргумента, а при переопределении унарных операторов операторная функция имеет один аргумент. Аргументы разделяются запятой,

а перед каждым аргументом указывается тип или класс, к которому принадлежит аргумент. Программный код операторной функции указывается в фигурных скобках. Как и для обычных функций, код можно размещать сразу после прототипа функции или в ином месте программы.

В качестве иллюстрации рассмотрим небольшую программу, в которой для объектов класса `MComp`, который отождествляем с комплексными числами, переопределяется операция сложения (то есть переопределяется оператор `+`). Программный код приведен в листинге 10.1.

Листинг 10.1. Переопределение оператора сложения внешней функцией

```
#include<iostream>
using namespace std;
//Класс для реализации комплексных чисел:
class MComp{
public:
    double Re;
    double Im;
};
//Переопределение оператора сложения:
MComp operator+(MComp x, MComp y){
    MComp z;
    z.Re=x.Re+y.Re;
    z.Im=x.Im+y.Im;
    return z;}
int main(){
    MComp a,b,c;
    a.Re=1;
    a.Im=2;
    b.Re=2;
    b.Im=3;
    //Сложение объектов:
    c=a+b;
    cout<<"c.Re = "<<c.Re<<"\n";
    cout<<"c.Im = "<<c.Im<<"\n";
    return 0;
}
```

В программе создается класс `MComp`, у которого есть два открытых поля `Re` (действительная часть комплексного числа) и `Im` (мнимая часть комплексного числа). Таким образом, операцию сложения объектов класса `MComp` следует определить так, чтобы в результате получался объект, который принадлежит тому же самому классу `MComp`, значение его поля `Re` равно сумме значений полей исходных объектов, а значение поля `Im` определяется как сумма полей `Im` объектов, для которых вычисляется сумма.

С целью реализации такого алгоритма создаем операторную функцию с прототипом `MComp operator+(MComp x, MComp y)`. В качестве типа значения, который возвращается функцией, указываем `MComp` (результат функции есть объект класса `MComp`), а аргументами функции (то есть операндами, которые участвуют в сложении) также являются объекты класса `MComp`. В теле функции создается временный локальный объект `z` класса `MComp`. Именно этот объект будет возвращаться в качестве результата функции. Поля объекта `z` определяются как сумма соответствующих полей объектов-аргументов функции. Как отмечалось, объект `z` инструкцией `return z` возвращается в качестве результата.

Главный метод программы содержит инструкцию объявления трех объектов `a`, `b` и `c` класса `MComp`, причем для двух из них (объекты `a` и `b`) присваиваются значения полей. Третий объект `c` определяется как сумма первых двух, для чего в программе использована инструкция `c=a+b`. Далее используется проверка значений полей объекта `c`: значения полей выводятся на экран. Он имеет следующий вид:

```
c.Re = 3
c.Im = 5
```

Легко убедиться, что сложение двух объектов выполнено в соответствии с описанным выше правилом.

Остановимся более детально на команде `c=a+b`. В правой части оператора присваивания стоит сумма двух объектов, которые принадлежат к классу `MComp`. В соответствии с кодом операторной функцией, которая была создана для переопределения оператора сложения, в этом случае создается временный объект с полями, значение которых равняется сумме полей объектов-операндов. Под временный объект в памяти выделяется место, а существует этот объект до тех пор, пока функцией, которая создала этот объект, не будет возвращен результат (то есть упомянутый объект). В результате операции присваивания все поля объекта `c` устанавливаются равными соответствующим полям временного объекта, про который шла речь выше. После этого временный объект уничтожается. В результате объект `c` будет иметь поля, равные сумме значений таких полей объектов, которые являются операндами в операции сложения.

Перегрузка операторной функции

Таким образом, теперь в программе можно складывать не только числа (например, типов `double`, `float` или `int`), существует возможность вычислять сумму объектов класса `MComp`, который создавался как прототип комплексного числа. Тем не менее, хотя операция сложения действительно

го и комплексного числа с математической точки зрения является вполне естественной и четко определенной, при попытке сложить, например, переменную типа `double` и объект класса `MComp` возникнет ошибка, обусловленная тем, что операция сложения объектов таких типов не определена. Проблему можно решить, предусмотрев при переопределении оператора сложения и такую ситуацию. По большому счету, в данном случае речь идет про дополнительное переопределение оператора сложения. На языке операторных функций это означает перегрузку созданной в предыдущем случае операторной функции. После внесения необходимых изменений программный код будет выглядеть так, как показано в листинге 10.2.

Листинг 10.2. Перегрузка внешней операторной функции

```
#include<iostream>
using namespace std;
//Класс для реализации комплексных чисел:
class MComp{
public:
double Re;
double Im;
};
//Операторная функция для сложения двух объектов:
MComp operator+(MComp x, MComp y){
MComp z;
z.Re=x.Re+y.Re;
z.Im=x.Im+y.Im;
return z;}
//Операторная функция для сложения числа и объекта:
MComp operator+(double x, MComp y){
MComp z;
z.Re=x+y.Re;
z.Im=y.Im;
return z;}
int main(){
MComp a,b,c;
double x=1.5;
a.Re=1;
a.Im=2;
b.Re=2;
b.Im=3;
//Сложение объектов:
c=a+b;
cout<<"c.Re = "<<c.Re<<"\n";
cout<<"c.Im = "<<c.Im<<"\n";
//Сложение числа и объекта:
c=x+c;
```

```

cout<<"c.Re = "<<c.Re<<"\n";
cout<<"c.Im = "<<c.Im<<"\n";
return 0;
}

```

Среди существенных изменений следует отметить наличие дополнительного варианта для операторной функции `operator+`. В данном случае, кроме использованного ранее, для функции введен еще и прототип `MComp operator+(double x, MComp y)`, в соответствии с которым первым операндом при сложении может быть переменная базового типа `double`, в то время как второй операнд – объект класса `MComp`. В этом случае к действительной части второго операнда прибавляется первый операнд, а комплексная часть результата определяется комплексной частью второго операнда. В основном методе программы, как и ранее, объявляется три объекта класса `MComp` и переменная базового типа `double`. Кроме сложения объектов класса `MComp`, проверяется корректность операции сложения переменной типа `double` и объекта класса `MComp`. Результат выполнения программы имеет следующий вид:

```

c.Re = 3
c.Im =5
c.Re = 4.5
c.Im = 5

```

Следует иметь в виду, что, несмотря на корректность, например, команды `c=c+c` (см. листинг 10.2), команда `c=c+x` была бы некорректной, поскольку при повторном переопределении оператора сложения определено, что именно первый операнд принадлежит типу `double`, а не второй. Чтобы была корректной и команда `c=c+x`, необходимо дополнительно внести в программу блок наподобие следующего:

Листинг 10.3. Код операторной функции при перегрузке оператора сложения

```

MComp operator+(MComp y, double x){
    MComp z;
    z.Re=x+y.Re;
    z.Im=y.Im;
    return z;}

```

После добавления в программу такого варианта операторной функции порядок следования операндов значения иметь не будет: для каждого из двух вариантов (*объект плюс число* и *число плюс объект*) существует версия переопределения оператора сложения.

Абсолютно так же можно переопределить операции вычитания, умножения и деления комплексных чисел. Кроме означенных, среди операторов, которые можно переопределять, также операторы инкремента, декремента,

сокращенные формы арифметических операторов, а также оператор присваивания (и это далеко не полный список). Рассмотрим наиболее интересные случаи.

Переопределение унарных операторов внешними функциями

Разум наш видит многое, для чего у нас нет словесных обозначений.

Данте Алигьери

Операторы инкремента и декремента, кроме того, что являются унарными, имеют еще одну существенную особенность: у них есть префиксная и постфиксная формы. В C++ допускается не только переопределение этих операторов, но также и каждой их формы (префиксной и постфиксной) в отдельности.

Методы переопределения унарных операторов проиллюстрируем все на том же примере с комплексными числами, переопределив операторы инкремента и декремента так, чтобы в префиксной форме действие оператора сводилось к изменению на единицу действительной части комплексного числа, а в постфиксной форме оператор влияет на мнимую часть. Начнем с того, что в программный код, который рассматривался выше, добавим операторную функцию для переопределения оператора инкремента (в префиксной форме). Программный код операторной функции приведен в листинге 10.4.

Листинг 10.4. Код операторной функции для переопределения оператора инкремента в префиксной форме

```
MComp operator++(MComp &x) {
    x.Re++;
    return x;}
```

Программный код достаточно простой, однако есть несколько деликатных моментов, на которые следует обратить внимание. Во-первых, поскольку оператор унарный, операнд всего один и, как следствие, у операторной функции всего один аргумент. Во-вторых, в результате действия оператора инкремента изменяется сам объект, поэтому он передается в функцию не по значению, а через ссылку, о чем свидетельствует инструкция & при объявлении аргумента. В теле операторной функции значение поля Re объекта, который передан (через ссылку!) функции в качестве аргумента, увеличивается на единицу, а сам этот объект возвращается в качестве результата. Наконец, отметим, что по умолчанию принимается, что указанным способом

определяется префиксная форма оператора инкремента. Для переопределения постфиксной формы оператора инкремента используем операторную функцию, программный код которой приведен в листинге 10.5.

Листинг 10.5. Код операторной функции для переопределения оператора инкремента в постфиксной форме

```
MComp operator++(MComp &x, int unused){
    x.Im++;
    return x;}
```

По сравнению с предыдущим случаем есть две особенности. Так, у функции появился второй целочисленный аргумент (инструкция `int unused`). Этот аргумент является формальным, при расчетах не используется, и единственное его предназначение в том, чтобы служить признаком переопределения постфиксной формы оператора. Кроме того, теперь на единицу увеличивается не поле `Re`, а поле `Im`, что соответствует увеличению на единицу мнимой части комплексного числа.

Таким же образом переопределяем оператор декремента, обе его формы. Переопределим также оператор `!` так, чтобы можно было вычислять комплексно-сопряженное число (напомним, что комплексно-сопряженным к числу $z = x + iy$ называется число $\bar{z} = x - iy$, то есть для вычисления комплексно-сопряженного числа следует умножить мнимую часть на -1). Операторная функция для переопределения оператора `!` имеет достаточно несложный программный код:

Листинг 10.6. Переопределение оператора для расчета комплексно-сопряженного числа

```
MComp operator!(MComp &x){
    x.Im*=-1;
    return x;}
```

Обращаем внимание на то, что в результате применения оператора `!` меняется объект-операнд. Поэтому если быть более корректным, то в данном случае речь идет не про расчет комплексно-сопряженного числа, а про замену исходного числа на комплексно-сопряженное. Чтобы в результате операции возвращалось комплексно-сопряженное число, без изменения исходного числа, необходимо в аргументе удалить инструкцию `&` передачи объекта через ссылку.

В листинге 10.7 приведен полный код программы, в которой использованы операторные функции для переопределения операций сложения, вычитания, инкремента, декремента и расчета комплексно-сопряженного числа.

Листинг 10.7. Программа с переопределением операций сложения, вычитания, инкремента, декремента и расчета комплексно-сопряженного числа

```

#include<iostream>
using namespace std;
//Класс для реализации комплексных чисел:
class MComp{
public:
double Re;
double Im;
};
//Операторная функция для сложения двух объектов:
MComp operator+(MComp x, MComp y){
MComp z;
z.Re=x.Re+y.Re;
z.Im=x.Im+y.Im;
return z;}
//Операторная функция для вычитания двух объектов:
MComp operator-(MComp x, MComp y){
MComp z;
z.Re=x.Re-y.Re;
z.Im=x.Im-y.Im;
return z;}
//Операторная функция для сложения числа и объекта:
MComp operator+(double x, MComp y){
MComp z;
z.Re=x+y.Re;
z.Im=y.Im;
return z;}
//Операторная функция для сложения объекта и числа:
MComp operator+(MComp y, double x){
MComp z;
z.Re=x+y.Re;
z.Im=y.Im;
return z;}
//Операторная функция для вычитания из числа объекта:
MComp operator-(double x, MComp y){
MComp z;
z.Re=x-y.Re;
z.Im=-y.Im;
return z;}
//Операторная функция для вычитания из объекта числа:
MComp operator-(MComp y, double x){
MComp z;
z.Re=y.Re-x;
z.Im=y.Im;

```

```

    return z;}
//Операторная функция для префиксной формы инкремента:
MComp operator++(MComp &x) {
    x.Re++;
    return x;}
//Операторная функция для постфиксной формы инкремента:
MComp operator++(MComp &x,int unused){
    x.Im++;
    return x;}
//Операторная функция для префиксной формы декремента:
MComp operator--(MComp &x) {
    x.Re--;
    return x;}
//Операторная функция для постфиксной формы декремента:
MComp operator--(MComp &x,int unused){
    x.Im--;
    return x;}
//Операторная функция для вычисления комплексно-сопряженного:
MComp operator!(MComp &x) {
    x.Im*=-1;
    return x;}
int main(){
    MComp a,b,c;
    double x=1.5, y=2.5;
    a.Re=1;
    a.Im=2;
    b.Re=2;
    b.Im=3;
    cout<<"*****";
    cout<<"\na.Re = "<<a.Re<<"\na.Im = "<<a.Im<<"\n";
    cout<<"*****";
    cout<<"\nb.Re = "<<b.Re<<"\nb.Im = "<<b.Im<<"\n";
    cout<<"*****";
    cout<<"\nx = "<<x<<"\ny = "<<y<<"\n";
    cout<<"*****";
    //Сложение объектов:
    c=a+b;
    cout<<"\nc=a+b: \n";
    cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
    cout<<"*****";
    //Сложение числа и объекта:
    c=x+c;
    cout<<"\nc=x+c: \n";
    cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
    cout<<"*****";
    //Сложение объекта и числа:

```

```

c=c+y;
cout<<"\nc=c+y: \n";
cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
cout<<"*****";
//Инкремент в префиксной форме:
++c;
cout<<"\n++c: \n";
cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
cout<<"*****";
//Инкремент в постфиксной форме:
c++;
cout<<"\nc++: \n";
cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
cout<<"*****";
//Вычитание объектов:
c=a-b;
cout<<"\nc=a-b: \n";
cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
cout<<"*****";
//Вычитание из числа объекта:
c=x-c;
cout<<"\nc=x-c: \n";
cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
cout<<"*****";
//Вычитание из объекта числа:
c=c-y;
cout<<"\nc=c-y: \n";
cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
cout<<"*****";
//Декремент в префиксной форме:
--c;
cout<<"\n--c: \n";
cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
cout<<"*****";
//Декремент в постфиксной форме:
c--;
cout<<"\nc--: \n";
cout<<"c.Re = "<<c.Re<<"\nc.Im = "<<c.Im<<"\n";
cout<<"*****";
//Комплексно-сопряженное:
!a;
cout<<"\n!a: \n";
cout<<"a.Re = "<<a.Re<<"\na.Im = "<<a.Im<<"\n";
cout<<"*****\n";
return 0;
}

```

Результат выполнения программы имеет следующий вид:

```

*****
a.Re = 1
a.Im = 2
*****
b.Re = 2
b.Im = 3
*****
x = 1.5
y = 2.5
*****
c=a+b:
c.Re = 3
c.Im = 5
*****
c=x+c:
c.Re = 4.5
c.Im = 5
*****
c=c+y:
c.Re = 7
c.Im = 5
*****
++c:
c.Re = 8
c.Im = 5
*****
c++:
c.Re = 8
c.Im = 6
*****
c=a-b:
c.Re = -1
c.Im = -1
*****
c=x-c:
c.Re = 2.5
c.Im = 1
*****
c=c-y:
c.Re = 0
c.Im = 1
*****
--c:
c.Re = -1
c.Im = 1

```

```

*****
c--:
c.Re = -1
c.Im = 0
*****
!a:
a.Re = 1
a.Im = -2
*****

```

Обращаем внимание на способ переопределения оператора вычитания `-`. Как и в случае переопределения оператора сложения, предусмотрены три ситуации: когда операндами являются объекты класса `MComp`, первый операнд типа `double` и второй операнд – объект класса `MComp`, и наоборот. Это позволяет вычитать из действительных чисел комплексные, и из комплексных – действительные.

Перегрузка операторов методами класса

Операторная функция, с помощью которой перегружается оператор, может быть внутренней, то есть являться методом класса. Принципиально ситуация мало отличается от рассмотренных ранее, однако есть ряд особенностей, главная из которых связана со способом передачи аргументов такой функции.

Строго говоря, драматические изменения происходят только с количеством аргументов в операторных функциях – при перегрузке бинарных операторов функции передается один аргумент, а при перегрузке унарных операторов функции аргументы не передаются (за исключением, пожалуй, перегрузки префиксной формы операторов инкремента и декремента). Причина связана с тем, что один аргумент внутренней операторной функции – объект, из которого вызывается эта функция. Доступ к этому объекту, в случае необходимости, можно получить через указатель `this`. В качестве иллюстрации рассмотрим, как изменится процедура перегрузки оператора сложения из рассмотренных выше примеров, если перегрузка осуществляется методом класса. Программный код приведен в листинге 10.8.

Листинг 10.8. Переопределение оператора сложения методом класса

```

#include<iostream>
using namespace std;
//Класс для реализации комплексных чисел:
class MComp{
public:

```

```

double Re;
double Im;
//Переопределение оператора сложения методом класса:
MComp operator+(MComp y) {
    MComp z;
    z.Re=Re+y.Re;
    z.Im=Im+y.Im;
    return z;}
};
int main() {
    MComp a,b,c;
    a.Re=1;
    a.Im=2;
    b.Re=2;
    b.Im=3;
    //Сложение объектов:
    c=a+b;
    cout<<"c.Re = "<<c.Re<<"\n";
    cout<<"c.Im = "<<c.Im<<"\n";
    return 0;
}

```

По сравнению с программным кодом в листинге 10.1 операторная функция, переопределяющая оператор сложения, внесена в описание класса. При этом она потеряла один аргумент – первый операнд в операции сложения объектов отождествляется с объектом, из которого вызывается операторная функция. Ссылки на поля этого объекта выполняются в формате `Re` и `Im`. При этом с точки зрения функциональности ничего не меняется – результат выполнения программы такой же, как и при выполнении программы из листинга 10.1.

Легко перегружаются с помощью методов класса унарные операторы. Например, в листинге 10.9 приведен программный код класса с переопределением оператора инкремента (в префиксной и постфиксной формах).

Листинг 10.9. Класс с переопределением оператора инкремента в префиксной и постфиксной формах

```

//Класс для реализации комплексных чисел:
class MComp{
public:
    double Re;
    double Im;
    //Переопределение префиксной формы инкремента:
    MComp operator++() {
        Re++;
        return *this;}
}

```

```
//Переопределение постфиксной формы инкремента:
MComp operator++(int unused){
    Im++;
    return *this;}
};
```

При переопределении префиксной формы оператора инкремента аргументы методу не передаются, а в качестве результата возвращается вызвавший метод объект (предварительно поле *Re* этого объекта увеличивается на единицу). При переопределении постфиксной формы оператора инкремента методу передается один целочисленный формальный аргумент (он нужен как индикатор переопределяемой формы оператора инкремента). Во всем остальном этот метод особенностей не имеет.

Перегрузка оператора присваивания

А ну снимай сапоги! Власть переменилась!

Из к/ф «Свадьба в Малиновке»

Как известно, если объекты принадлежат одному классу, к ним можно применять операцию копирования. В этом случае по умолчанию выполняется побитовое копирование: объект, указанный слева от оператора присваивания = получает значения полей объекта, указанного справа от оператора присваивания. Хотя в большинстве случаев такая ситуация вполне приемлема, все же нередко приходится переопределять процедуру копирования объектов. Реализуется это посредством перегрузки оператора присваивания.

В принципе, перегрузка оператора присваивания выполняется так же, как и прочих бинарных операторов. Однако оператор присваивания нельзя перегружать внешней функцией. Поэтому при перегрузке оператора присваивания определяется метод класса. Кроме того, следует помнить, что в качестве значения в методе перегрузки оператора присваивания возвращается объект, из которого вызывается метод. Пример перегрузки оператора присваивания приведен в листинге 10.10.

Листинг 10.10. Переопределение оператора присваивания

```
#include<iostream>
using namespace std;
//Описание класса:
class MComp{
public:
    double Re;
    double Im;
```



```

//Переопределение оператора присваивания:
MComp operator=(MComp x) {
    Re=x.Re+1;
    Im=x.Im-1;
    return *this;}
};

int main(){
    MComp a,b;
    a.Re=1;
    a.Im=2;
    //Присваивание объектов:
    b=a;
    cout<<"b.Re = "<<b.Re<<"\n";
    cout<<"b.Im = "<<b.Im<<"\n";
    return 0;
}

```

Оператор присваивания переопределен так, что при присваивании одного объекта другому значение поля Re при копировании увеличивается на единицу, а значение поля Im уменьшается на единицу. При этом значения полей присваиваемого объекта (того, что слева от оператора присваивания) остаются неизменными. В результате выполнения программы получим следующее:

```

b.Re = 2
b.Im = 1

```

Кроме перечисленных выше операторов, переопределяются также сокращенные операторы присваивания, операторы [], (), -, -, оператор , , а также операторы new и delete.

Примеры решения задач

В представленных здесь примерах широко используются операторные функции. Некоторые примеры иллюстрируют альтернативные способы решения задач, уже рассматривавшихся ранее.

■ Индексирование объектов

Оператор [] (квадратные скобки) может перегружаться так же, как и рассмотренные выше арифметические операторы. Здесь рассмотрим пример, в котором посредством перегрузки оператора [] решается задача по индексированию объектов. В частности, в программе описывается класс с полем-целочисленным массивом. Обычный способ обращения к элементу массива-члена класса подразумевает указание объекта, имени поля-

массива и индекса элемента массива. Переопределив оператор `[]`, сможем обращаться к элементу поля-массива, указав после имени объекта соответствующий индекс. Программный код приведен в листинге 10.11.

Листинг 10.11. Индексирование объектов

```
#include <iostream>
#include <cstdlib>
using namespace std;
//Размер полей-массивов:
const int n=10;
//Класс с полем-массивом:
class RealNums{
public:
    //Поле-массив:
    int p[n];
    //Конструктор класса:
    RealNums(){
        int k;
        for(k=0;k<n;k++)
            p[k]=rand()%n;
    }
    //Перегрузка оператора []:
    int &operator[](int i){
        return p[i];
    }
    //Перегрузка оператора +:
    RealNums operator+(RealNums obj){
        int i;
        RealNums tmp;
        for(i=0;i<n;i++)
            tmp[i]=p[i]+obj[i];
        return tmp;}
    //Метод для вывода значений массива:
    void show(){
        int i;
        for(i=0;i<n;i++)
            printf("%3d",p[i]);
        cout<<endl;
    }
};

int main(){
    RealNums a,b;
    a.show();
    b.show();
    (a+b).show();
```

```

for(int i=0;i<n;i++) a[i]=b[i]-a[i];
a.show();
return 0;}

```

В программе объявляется глобальная целочисленная константа `n`, определяющая размер поля-массива `p` создаваемого класса `RealNums`. У класса есть конструктор, с помощью которого при создании элементы массива-поля соответствующего объекта заполняются случайными числами. В классе, как отмечалось, перегружается оператор `[]`. Оператор является бинарным, поэтому для него указан целочисленный аргумент – этот аргумент соответствует индексу, который указывается в квадратных скобках. Второй операнд оператора `[]` – объект, для которого выполняется индексирование (объект, после которого указываются квадратные скобки). Для того чтобы индексированный объект можно было использовать в операциях присваивания в левой части (чтобы объектам с индексом можно было присваивать значения), соответствующая операторная функция возвращает не значение элемента массива (значение типа `int`), а ссылку на этот элемент, о чем свидетельствует инструкция `&` в прототипе операторной функции.

Помимо операторной функции для перегрузки оператора `[]`, в классе определена операторная функция для перегрузки оператора сложения. В результате сложения двух объектов получаем объект того же класса, элементами поля-массива которого являются суммы соответствующих элементов складываемых объектов. Обращаем внимание, что в этой операторной функции используется индексирование объекта-аргумента (инструкция `obj[i]` эквивалентна инструкции `obj.p[i]`).

Метод `show()` класса `RealNums` используется для вывода значений элементов массива `p` объекта на экран.

В главном методе программы создается два объекта, а их содержимое выводится на экран. Далее посредством перегруженного оператора сложения определяется результат сложения двух массивов, а разность второго и первого массивов вычисляется поэлементно в рамках оператора цикла, при этом используется индексация объектов. Результат выполнения программы может иметь следующий вид:

1	7	4	0	9	4	8	8	2	4
5	5	1	7	1	1	5	2	7	6
6	12	5	7	10	5	13	10	9	10
4	-2	-3	7	-8	-3	-3	-6	5	2

Отметим, что рассмотренный выше способ перегрузки оператора `[]` является далеко не единственным. Например, можно перегрузить этот оператор так, чтобы при индексации объектов автоматически выполнялась и обрабатывалась ситуация по выходу индексной переменной за пределы массива.

■ Скалярное и векторное произведение векторов

Решим задачу о вычислении скалярного и векторного произведения двух векторов, для чего определим специальный класс и перегрузим ряд операторов. В листинге 10.12 приведен соответствующий программный код.

Листинг 10.12. Скалярное и векторное произведение векторов

```
#include <iostream>
using namespace std;
//Класс для реализации векторов:
class Vector{
    //Координаты вектора - закрытый массив-член класса:
    double coords[3];
public:
    //Перегрузка оператора [] для индексации объектов:
    double &operator[](int i){
        int k=i%3;
        return coords[k];}
    //Перегрузка оператора () для присваивания координатам
    //значений:
    Vector operator()(double x,double y,double z){
        coords[0]=x;
        coords[1]=y;
        coords[2]=z;
        return *this;}
    //Перегрузка оператора () для вычисления векторного
    //произведения:
    Vector operator()(Vector a,Vector b){
        for(int i=0;i<3;i++){
            coords[i]=a[i+1]*b[i+2]-a[i+2]*b[i+1];
        }
        return *this;}
    //Перегрузка оператора * для вычисления скалярного
    //произведения:
    double operator*(Vector obj){
        double res=0;
        for(int i=0;i<3;i++) res+=coords[i]*obj[i];
        return res;}
    //Конструктор с тремя аргументами:
    Vector(double x,double y,double z){
        coords[0]=x;
        coords[1]=y;
        coords[2]=z;}
    //Метод для отображения координат вектора:
```

```

void show() {
    cout<<" ("<<coords[0]<<" , "<<coords[1]<<" ,
                                     "<<coords[2]<<") \n"; }

//Конструктор без аргументов:
Vector() {
    for(int i=0;i<3;i++)
        coords[i]=0; }
};

int main() {
    //Объявление трех векторов:
    Vector a,b(2,1,3),c;
    //Определение координат вектора:
    a(1,-3,4);
    //Отображение координат векторов:
    a.show();
    b.show();
    //Скалярное произведение:
    cout<<"a*b="<<a*b<<endl;
    //Векторное произведение:
    c(a,b);
    cout<<"[ab]=";
    c.show();
    return 0;}

```

Для реализации векторов создается класс с названием `Vector`. Компоненты вектора являются элементами массива `coords` размера 3 (тип элементов `double`). Массив является закрытым полем класса `Vector`, поэтому напрямую вне класса доступ к элементам массива получить не удастся. Для доступа к элементам массива перегружается оператор `[]`. Помимо возможности индексировать объекты, т.е. получать доступ к элементам массива, указывая соответствующий индекс для объекта, в данном случае этот оператор перегружен так, что предотвращается возможность выхода за пределы массива. Достигается это путем циклического перехода при превышении индексом допустимого значения. Например, если указан индекс 3, (максимально допустимый индекс для массива размера 3 равен 2), то возвращаться будет элемент с индексом 0. Значение 4 соответствует индексу 1 и так далее. Такое переопределение оператора `[]` позволит в дальнейшем достаточно просто реализовать процесс вычисления векторного произведения.

Оператор умножения `*` перегружается в классе для вычисления скалярного произведения. Напомним, что скалярным произведением векторов

$\vec{a} = (a_1, a_2, a_3)$ и $\vec{b} = (b_1, b_2, b_3)$ называется число $\vec{a}\vec{b} = \sum_{k=1}^3 a_k b_k$. При

перегрузке оператора умножения у соответствующей операторной функции

указывается один аргумент-объект класса `Vector` (второй операнд, а первым операндом является объект, из которого вызывается операторная функция). В теле операторной функции используется индексация объектов, которая возможна благодаря перегрузке оператора `[]`.

Оператор `()` перегружается дважды. Здесь следует отметить следующее обстоятельство. У оператора может быть произвольное число аргументов. Это те операнды, которые при вызове оператора указываются в круглых скобках (разделяются операнды запятыми). Первый способ перегрузки оператора `()` подразумевает наличие трех (помимо объекта, из которого вызывается операторная функция) аргументов типа `double`. В этом случае аргументы операторной функции присваиваются в качестве значений элементам поля-массива объекта. Другими словами, если в программе после имени объекта класса `Vector` указать в круглых скобках три числовых значения, эти значения будут присвоены координатам вектора, с которым отождествляется объект.

Второй способ перегрузки оператора `()` определяется для двух аргументов-объектов класса `Vector`. В этом случае вычисляется векторное произведение векторов-аргументов операторной функции. Результат векторного произведения (это вектор) записывается в объект, из которого вызывается операторная функция. Например, если a , b и c – объекты класса `Vector`, то в результате выполнения команды `c(a, b)` объект `c` будет соответствовать векторному произведению векторов, представленных объектами a и b . Результатом векторного произведения векторов $\vec{a} = (a_1, a_2, a_3)$ и $\vec{b} = (b_1, b_2, b_3)$ является вектор $\vec{c} \equiv [\vec{a}\vec{b}]$ с компонентами $c_1 = a_2b_3 - a_3b_2$, $c_2 = a_3b_1 - a_1b_3$ и $c_3 = a_1b_2 - a_2b_1$. С учетом цикличности индексов эти зависимости могут быть формально представлены как $c_k = a_{k+1}b_{k+2} - a_{k+2}b_{k+1}$, где индекс компонентов векторов $k = 1, 2, 3$. Здесь стоит обратить внимание, что при работе с соответствующими объектами индексация начинается не с 1, а с 0.

Помимо этого в классе описан перегруженный конструктор с тремя аргументами и без аргументов. В первом случае при создании объекта ему передаются значения координат соответствующего вектора. Во втором случае создается объект для нулевого вектора – все компоненты вектора нулевые. Метод `show()` предназначен для отображения значений компонент вектора.

В главном методе программы создается три объекта a , b и c класса `Vector` – два (объекты a и c) соответствуют нулевым векторам и один (объект b) соответствует вектору с ненулевыми компонентами. Далее командой `a(1, -3, 4)` изменяются значения элементов поля-массива объекта a . Значения элементов массивов для объектов a и b выводятся с помощью метода `show()`. Скалярное произведение вычисляется командой `a*b`, а вектор-

ное – командой `c(a, b)`. Результат выводится на экран. В итоге получаем следующее:

```
(1, -3, 4)
(2, 1, 3)
a*b=11
[ab]=(-13, 5, 7)
```

Здесь хочется сделать еще одно замечание, касающееся индексации объектов. Оператор `[]` можно переопределить так, что индексация объектов будет начинаться с 1, как в случае с индексами вектора, а не с 0. Такое не-сложное задание читатель может решить самостоятельно.

■ Операции с матрицами

Путем перегрузки базовых операторов существенно упрощается процесс работы с такими математическими структурами, как матрицы. Здесь создадим специальный программный код, в котором предусмотрим возможность выполнять некоторые наиболее простые операции с матрицами, а именно складывать матрицы и умножать их. Более точно, решим следующие задачи. Создадим класс для работы с квадратными матрицами. Элементы матрицы реализуются в виде двумерного массива-поля класса. Перегрузим оператор `[]` так, чтобы можно было индексировать объекты – теперь используется два индекса, поскольку соответствующее поле объекта является двумерным массивом. Для вычисления суммы матриц перегрузим оператор сложения, а для вычисления произведения матриц перегрузим оператор умножения. Программный код приведен в листинге 10.13.

Листинг 10.13. Операции с матрицами

```
#include <iostream>
using namespace std;
//Размер квадратной матрицы:
const int n=3;
//Класс для реализации матриц:
class Matrix{
public:
    //Поле - двумерный массив:
    int matr[n][n];
    //Перегрузка оператора []:
    int *operator[](int k){
        return matr[k];}
    //Перегрузка оператора сложения +:
    Matrix operator+(Matrix obj){
        Matrix tmp(0);
```

```

        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                tmp[i][j]=matr[i][j]+obj[i][j];
        return tmp;}

//Перегрузка оператора умножения *:
Matrix operator*(Matrix obj){
Matrix tmp(0);
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
        for(int k=0;k<n;k++)
            tmp[i][j]+=matr[i][k]*obj[k][j];
    return tmp;}

//Метод для отображения значений матрицы:
void show(){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            printf("%3d",matr[i][j]);
        }
        cout<<endl;
    }
}

//Конструктор с аргументом:
Matrix(int k){
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            matr[i][j]=k;
}

//Конструктор без аргументов:
Matrix(){
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            matr[i][j]=rand()%5-2;
}

};

int main(){
    //Создание объектов:
    Matrix A,B,C(0),D(0);
    cout<<"Matrix A:\n";
    A.show();
    cout<<"Matrix B:\n";
    B.show();
    //Сумма матриц:
    cout<<"Matrix C=A+B:\n";
    C=A+B;
    C.show();
    //Произведение матриц:
    cout<<"Matrix D=A*B:\n";

```



```

D=A*B;
D.show();
return 0;}

```

В первую очередь стоит обратить внимание на способ перегрузки оператора `[]`. В данном случае индексируемые объекты имеют два индекса. Поэтому формально перегружается только процедура индексирования по первому индексу. Например, предположим, что объект `obj` относится к классу `Matrix`. Инструкция вида `obj[i][j]` должна предоставлять доступ к элементу `matr[i][j]` поля этого объекта. Оператор `[]` перегружается так, что в результате вызова соответствующей операторной функции (с аргументом `i`) возвращается указатель `matr[i]`. О том, что возвращается указатель, свидетельствует инструкция `*` в прототипе операторной функции. Поэтому в принципе перегрузка оператора `[]` сводится к тому, что инструкция вида `obj[i]` является способом обращения к указателю `matr[i]`. Наличие вторых прямоугольных скобок с индексом после этой инструкции означает индексирование указателя.

Как известно, при вычислении суммы матриц в результате получаем матрицу, элементы которой равны сумме соответствующих элементов исходных матриц. Так, если матрица A имеет элементы a_{ij} , а матрица B – элементы b_{ij} (индексы $i, j = 1, 2, \dots, n$), то матрица $C = A + B$ имеет элементы $c_{ij} = a_{ij} + b_{ij}$. Элементы матрицы $D = A * B$ определяются как $d_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$. Именно эти соотношения использовались при перегрузке операторов сложения и умножения соответственно.

Кроме перегрузки операторов, в классе предусмотрен перегруженный конструктор. Если конструктор вызывается без аргументов, двумерный массив объекта заполняется случайными числами в диапазоне от -2 до 2 включительно. Если конструктору передан целочисленный аргумент, все элементы двумерного массива объекта заполняются соответствующим значением. Метод `show()` используется для вывода значений массива на экран. В главном методе программы проверяется корректность работы перегруженных операторов. Результат выполнения программы может иметь следующий вид:

```

Matrix A:
-1    0    2
-2    2    2
 1    1    0
Matrix B:
 2   -2   -2
-1    0   -1
-1   -2    0

```

```

Matrix C=A+B:
  1   -2   0
 -3    2   1
  0   -1   0
Matrix D=A*B:
 -4   -2   2
 -8    0   2
  1   -2  -3

```

Для практического использования этого кода разумно также выполнить дополнительную перегрузку операторов так, чтобы как минимум можно было умножать матрицы на число и вычислять разность матриц.

■ Вектор-функция

Под вектор-функцией в данном случае будем подразумевать такую функцию, которая в качестве результата возвращает массив значений. Как известно, обычная функция своим результатом может возвращать все, кроме массива. Поэтому для определения вектор-функции создадим специальный класс, в котором перегрузим ряд операторов (в первую очередь оператор `[]`), чтобы формально результат вызова функции можно было индексировать. Разумеется, о создании реальной вектор-функции речь не идет, но внешне иллюзия наличия вектор-функции будет полной. Для простоты рассмотрим ситуацию, когда вектор-функция описывает некоторую пространственную силу в трехмерном декартовом пространстве. Это означает, что вектор-функция в качестве значения возвращает массив размера 3. Каждый из трех компонентов силы является функцией времени. Соответствующий программный код приведен в листинге 10.14.

Листинг 10.14. Вектор-функция

```

#include <iostream>
#include <cmath>
using namespace std;
const double pi=3.1415;
class MyFunc{
    double f[3];
    double t;
    double f1(double z){
        return 3*sin(6*pi*z)*cos(2*pi*z);
    }
    double f2(double z){
        return 5*sin(6*pi*z)*sin(2*pi*z);
    }
    double f3(double z){
        return 4*cos(6*pi*z);
    }
    void set(){

```

```

    f[0]=f1(t);
    f[1]=f2(t);
    f[2]=f3(t);}
public:
double *operator()(double z){
    t=z;
    set();
    return f;}
double operator[](int k){
    if(!(k%4)) return t;
    else return f[(k-1)%4];}
MyFunc(){
    t=0;
    set();}
};

int main(){
    MyFunc F;
    double time=0.125;
    int i;
    for(i=0;i<3;i++)
        cout<<F(time)[i]<<" ";
    cout<<endl;
    for(i=1;i<=4;i++)
        cout<<F[i]<<" ";
    cout<<endl;
    return 0;}

```

Основу программного кода составляет класс `MyFunc`. У класса есть закрытое поле-массив `f` размера 3. Через этот массив реализуется результат вычисления вектор-функции. Аргумент вектор-функции записывается в закрытое поле `double t`. Закрытые методы `f1()`, `f2()` и `f3()` определяют функциональные зависимости для компонентов вектор-силы. Используются следующие зависимости $f_1(t) = 3 \sin(6\pi t) \cos(2\pi t)$, $f_2(t) = 5 \sin(6\pi t) \sin(2\pi t)$ и $f_3(t) = 4 \cos(6\pi t)$. Закрытый метод `set()` используется для заполнения массива `f` на основе значения поля `t`.

Оператор `()` перегружается так, что в полю `t` присваивается значение, указанное в скобках, а затем на основании этого значения заполняется массив `f`. В качестве результата операторной функции возвращается указатель на данный массив. В силу этого обстоятельства результат вызова операторной функции можно индексировать. Другими словами, чтобы узнать значение i -го компонента вектор-функции, нужно использовать команду вида `obj(t)[i]`, где `obj` – объект класса `MyFunc`.

При перегрузке оператора `[]` сразу решалось несколько задач. Во-первых, если в квадратных скобках указан нулевой индекс, возвращается значение

аргумента вектор-функции – т.е. значение поля `t` соответствующего объекта. Во-вторых, индексация компонентов вектор-функции начинается с 1 (диапазон изменения индекса от 1 до 3 включительно). В-третьих, используется принцип циклической индексации объектов (индекс 4 соответствует индексу 0, индекс 5 соответствует индексу 1 и т.д.).

У класса есть конструктор, которым вычисляется значение вектор-функции в начальный момент, а полю `t` соответственно присваивается нулевое значение.

В главном методе программы создается объект класса `MyFunc` и для предопределенного момента времени `t=0.125` различными способами вычисляется значение вектор-функции. Результат выполнения программы имеет вид:

```
1.50014 2.50012 -2.82823
1.50014 2.50012 -2.82823 0.125
```

Обращаем внимание, что значение компонента вектор-функции `F` (объект класса `MyFunc`) можно вычислить либо командой вида `F(t)[i]`, либо `F[i]`. Причем в первом случае индексация (для компонентов силы) начинается с 0, а во втором – с 1 (командой `F[0]` возвращается момент времени `t`).

■ Операции с полиномами

Рассмотрим программу, предназначенную для обработки полиномиальных выражений. Поскольку полином (как функциональная зависимость) однозначно определяется набором коэффициентов, его разумно реализовать в виде класса с полем-массивом числовых значений. Эти числовые значения отождествим с коэффициентами полинома. В качестве операций, выполняемых над полиномом, рассмотрим сложение полиномов, вычитание полиномов, умножение и деление полинома на число, а также вычисление производных от полинома. При этом максимально возможную степень полинома будем предполагать известной (что позволит нам определить соответствующее значение в виде целочисленной константы).

Выполнять перечисленные выше операции будем посредством перегрузки базовых операторов. Программный код приведен в листинге 10.15.

Листинг 10.15. Операции с полиномами

```
#include <iostream>
#include <cmath>
using namespace std;
//Степень полинома:
const int n=5;
class Polynom{
```

```

public:
//Коэффициенты полинома:
double a[n+1];
//Перегрузка оператора "скобки" (т.е. ()):
double operator()(double z){
    int i;
    double s=0;
    for(i=0;i<=n;i++) s+=a[i]*pow(z,i);
    return s;}
//Перегрузка оператора "запятая" (т.е. ,):
Polynom operator,(int k){
    int i;
    Polynom tmp;
    for(i=0;i<=n;i++) tmp.a[i]=a[i];
    switch(k){
        case 0:
            break;
        case 1:
            for(i=0;i<n;i++)
                tmp.a[i]=tmp.a[i+1]*(i+1);
            tmp.a[n]=0;
            break;
        default:
            //Рекурсия:
            tmp=(tmp,1,k-1);}
    return tmp;}
//Перегрузка оператора сложения:
Polynom operator+(Polynom obj){
    Polynom tmp;
    for(int i=0;i<=n;i++)
        tmp.a[i]=a[i]+obj.a[i];
    return tmp;}
//Перегрузка оператора вычитания:
Polynom operator-(Polynom obj){
    Polynom tmp;
    for(int i=0;i<=n;i++)
        tmp.a[i]=a[i]-obj.a[i];
    return tmp;}
//Перегрузка оператора умножения:
Polynom operator*(double b){
    Polynom tmp;
    for(int i=0;i<=n;i++)
        tmp.a[i]=a[i]*b;
    return tmp;}

```

```

//Перегрузка оператора деления:
Polynom operator/(double b){
    Polynom tmp;
    for(int i=0;i<=n;i++){
        tmp.a[i]=a[i]/b;
    }
    return tmp;}
//Конструктор класса:
Polynom(){
    for(int i=0;i<=n;i++){
        a[i]=i+1;
    }
};

//Внешняя функция для отображения коэффициентов полинома:
void show(Polynom obj){
    for(int i=0;i<=n;i++){
        cout<<obj.a[i]<<" ";
    }
    cout<<endl;
}

int main(){
    //Создание объектов:
    Polynom obj1,obj2;
    //Переменные для полиномов:
    double x=1,y=-1;
    //Коэффициенты полинома:
    show(obj1);
    //Деление на число:
    obj1=obj1/2;
    show(obj1);
    //Умножение на число:
    obj1=obj1*2;
    show(obj1);
    //Значение полинома:
    cout<<obj1(x)<<endl;
    //Производные от полинома:
    for(int i=0;i<=3;i++){
        obj2=(obj1,i);
        show(obj2);
    }
    //Значение производной:
    cout<<(obj1,1)(y)<<endl;
    //Значение суммы полиномов:
    cout<<(obj1+obj2)(x)<<endl;
    //Значение разности полиномов:
    cout<<(obj1-(obj1,1,2))(y)<<endl;
    return 0;}

```

Степень полинома задается глобальной константой `int n` (в данном случае 5). В классе `Polynom` объявляется массив `a` размера `n+1`. Элементы этого массива определяют коэффициенты полинома. Так, если a_k , $k = 0, 1, \dots, n$ – элементы полинома, то сам полином $P_n(x) = \sum_{k=0}^n a_k x^k$. Эти коэффициенты однозначно определяют структуру полинома. Конструктором класса при создании объекта массив заполняется последовательностью натуральных чисел. Другими словами все создаваемые объекты соответствуют полиному вида $P_n(x) = 1 + 2x + 3x^2 + \dots + (n+1)x^n$.

Достаточно тривиально перегружаются операторы сложения и вычитания: при сложении полиномов (объектов, через которые реализуются полиномы) необходимо сложить соответствующие элементы массивов объектов, а при вычитании элементы отнимаются.

В классе переопределены операторы умножения и деления. Первым операндом в соответствующих командах должен быть объект класса `Polynom`, второй операнд – действительное число. Умножение или деление полинома на число подразумевает умножение или деление на это число каждого элемента массива `a`.

Оператор «скобки» (т.е. `()`) перегружается так, чтобы значение полинома в точке можно было вычислить, указав после имени объекта класса `Polynom` в круглых скобках аргумент – действительное число.

Достаточно своеобразно перегружается оператор «запятая» (т.е. `,`). Это бинарный оператор. Первый его операнд – объект, стоящий слева от запятой (из этого объекта вызывается соответствующая операторная функция), а второй операнд – объект, стоящий справа от запятой. В данном случае перегружаем оператор для вычисления производной от полинома. Объект для полинома, от которого вычисляется производная, является первым операндом, а порядок производной (целое число) – второй операнд. Поэтому у операторной функции один аргумент целого типа. В теле операторной функции использован оператор `switch()`. Проверяется аргумент. При нулевом аргументе в качестве результата возвращается исходный полином. При единичном элементе в явном виде выписана процедура вычисления коэффициентов полинома-результата. Они вычисляются на основе коэффициентов исходного полинома. Коэффициенты для полинома-производной b_k вычисляются на основе коэффициентов исходного полинома a_k в соответствии с соотношением $b_k = (k+1)a_{k+1}$, при этом $b_n = 0$. Вычисление производных более высокого порядка определяется через рекурсию. Так, в операторной функции создается временный объект `tmp` класса `Polynom`. Этот объект возвращается в качестве результата и на начальном этапе совпадает с тем полиномом, для

которого вычисляется производная. Если аргумент функции k больше единицы (точнее, не равен нулю или единице), то результат вычисляется командой `tmp=(tmp, 1, k-1)` – производная порядка $k-1$ от первой производной от полинома `tmp`. Скобки в данном случае нужны, чтобы изменить приоритет оператора запятой по отношению к оператору присваивания.

Кроме класса `Polynom`, в программе описывается функция `show()` для отображения элементов массива объекта, указанного аргументом функции.

В главном методе программы проверяется работа перегруженных операторов. Результат выполнения программы будет иметь следующий вид:

```
1 2 3 4 5 6
0.5 1 1.5 2 2.5 3
1 2 3 4 5 6
21
1 2 3 4 5 6
2 6 12 20 30 0
6 24 60 120 0 0
24 120 360 0 0 0
18
525
-267
```

Обращаем внимание, что через запятую можно указывать несколько аргументов – главное, чтобы первый операнд принадлежал к классу `Polynom`. Так, команда `(obj1, 1, 2)` означает третью ($1+2=3$) производную от полинома, представленного объектом `obj1`.

Резюме

Из всего, что мне говорили, последнее, что я понял, было «Здравствуйте».

Дж. Буш (старший)

1. Операторная функция – функция, формальным названием которой является переопределенный оператор. Операторная функция может быть как внешней, так и методом класса.
2. Аргументами операторной функции являются операнды соответствующего выражения. В случае использования внешней операторной функции, для бинарных операторов аргументов два, для унарных – один. Если используется метод класса, бинарным операторам соответствует один аргумент, у унарных операторов аргументов нет. Унарные операторы инкремента и декремента имеют в этом отношении свои особенности.

3. По умолчанию при определении операторной функции для операторов инкремента и декремента подразумевается префиксная форма этих операторов. Для определения постфиксной формы этих операторов в качестве аргумента необходимо указать один формальный целочисленный аргумент.
4. Операторные функции могут перегружаться. Перегрузка операторных функций формально производится так же, как и перегрузка обычных функций, с поправкой на особенности синтаксиса объявления операторной функции.

Контрольные вопросы

*Я горячий друг истины, но не желаю
быть ее мучеником.*

Вольтер

1. Что такое операторная функция? В чем ее особенности?
2. Как определяется операторная функция?
3. Какие у операторной функции могут быть аргументы?
4. В чем принципиальная разница между внешними операторными функциями и операторными функциями-членами класса?
5. В чем особенности создания операторных функций для операторов инкремента и декремента?

Задачи для самостоятельного решения

В предлагаемых для самостоятельного решения задачах рекомендуется использовать переопределение операторных функций.

Задача 1. Написать программу с классом для реализации комплексных чисел в алгебраической форме (комплексное число $z = x + iy$, $\text{Re}(z) = x$ – действительная часть комплексного числа, $\text{Im}(z) = y$ – мнимая часть комплексного числа). Путем перегрузки операторов предусмотреть возможность выполнения следующих действий: сложение, вычитание, умножение и деление, причем одним из операндов может быть действительное число. Результат – комплексное число в алгебраическом представлении.

Задача 2. Написать программу с классом для реализации комплексных чисел в тригонометрической форме. В этом случае комплексное число $z = x + iy$ определяется модулем $r = \sqrt{x^2 + y^2}$ и аргументом φ ($\cos(\varphi) = x/r$ и $\sin(\varphi) = y/r$) и может быть представлено в виде $z = r \exp(i\varphi) = r \cos(\varphi) + ir \sin(\varphi)$. Путем перегрузки операторов предусмотреть возможность сложения и вычитания комплексных чисел, а также комплексного и действительного чисел. Результат – комплексное число в тригонометрическом представлении.

Задача 3. Написать программу с классами для реализации комплексных чисел в алгебраической и тригонометрической формах. Путем перегрузки операторов предусмотреть возможность возведения комплексного числа в тригонометрическом представлении в целочисленную степень. Результат – число в алгебраическом представлении.

Задача 4. Написать программу с классами для реализации комплексных чисел в алгебраической и тригонометрической формах. Путем перегрузки операторов предусмотреть возможность возведения комплексного числа в алгебраическом представлении в целочисленную степень. Результат – число в тригонометрическом представлении.

Задача 5. Написать программу с классами для реализации комплексных чисел в алгебраической и тригонометрической формах. Перегрузить операторы сложения и вычитания так, чтобы можно было складывать и вычитать комплексные числа, причем числа могут быть представлены в разной форме (реализованы через объекты разных классов). Результат имеет тип первого операнда.

Задача 6. Написать программу с классами для реализации комплексных чисел в алгебраической и тригонометрической формах. Перегрузить операторы умножения и деления так, чтобы можно было умножать и делить комплексные числа, причем числа могут быть представлены в разной форме (реализованы через объекты разных классов). Результат имеет тип первого операнда.

Задача 7. Написать программу с классом для реализации полинома. Коэффициенты полинома представить в виде поля-массива класса. Путем перегрузки операторов реализовать следующую операцию: в полиномиальном выражении сделать замену полиномиальной переменной $x = z + a$ (a – действительное число). Результатом является полином, записанный в терминах новой полиномиальной переменной z . Воспользовать-

ся тем, что $(a + z)^n = \sum_{k=0}^n C_n^k z^k a^{n-k}$, где биномиальные коэффициенты

$$C_n^k = \frac{n!}{k!(n-k)!}.$$

Задача 8. Написать программу с классом для реализации выражения вида $Q_n(x) = \sum_{k=0}^n \frac{a_k}{x^k} = a_0 + \frac{a_1}{x} + \dots + \frac{a_n}{x^n}$. Путем перегрузки операторов, предусмотреть возможность сложения, вычитания таких выражений, а также умножения и деления на число. Результат – объект того же класса, что и исходный.

Задача 9. Написать программу с классом для реализации выражения вида $Q_n(x) = \sum_{k=0}^n a_k \exp(-kx)$. Путем перегрузки операторов предусмотреть возможность сложения, вычитания таких выражений, а также умножения и деления на число. Результат – объект того же класса, что и исходный.

Задача 10. Написать программу с классом для реализации выражения вида $Q_n(x) = \sum_{k=0}^n a_k \exp(-kx)$. Путем перегрузки операторов предусмотреть возможность вычисления производной произвольного порядка. Результат – объект того же класса, что и исходный.

Задача 11. Написать программу с классом для реализации выражения вида $Q_n(x) = \sum_{k=0}^n (a_k \sin(kx) + b_k \cos(kx))$. Путем перегрузки операторов предусмотреть возможность сложения, вычитания таких выражений, а также умножения и деления на число. Результат – объект того же класса, что и исходный.

Задача 12. Написать программу с классом для реализации выражения вида $Q_n(x) = \sum_{k=0}^n (a_k \sin(kx) + b_k \cos(kx))$. Путем перегрузки операторов предусмотреть возможность вычисления производной произвольного порядка. Результат – объект того же класса, что и исходный.

Задача 13. Написать программу с классом для реализации выражения вида $Q_n(x) = \sum_{k=0}^n (a_k sh(kx) + b_k ch(kx))$. Путем перегрузки операторов предусмотреть возможность сложения, вычитания таких выражений, а также умножения и деления на число. Результат – объект того же класса, что и исходный.

Задача 14. Написать программу с классом для реализации выражения вида $Q_n(x) = \sum_{k=0}^n (a_k sh(kx) + b_k ch(kx))$. Путем перегрузки операторов предусмотреть возможность вычисления производной произвольного порядка. Результат – объект того же класса, что и исходный.

Задача 15. Написать программу с классом для реализации выражения вида $Q_n(x) = \sum_{k=1}^n \frac{a_k}{x^k} + \sum_{k=0}^n b_k x^k$. Путем перегрузки операторов предусмотреть возможность сложения, вычитания таких выражений, а также умножения и деления на число. Результат – объект того же класса, что и исходный.

Задача 16. Написать программу с классами для реализации полиномов степени n и $2n$. Путем перегрузки операторов предусмотреть возможность преобразования (точнее, создания на основе) объекта для полинома степени n в объект для полинома степени $2n$. Реализовать процесс вычисления произведения двух полиномов степени n (результат – полином степени $2n$).

Задача 17. Написать программу с классом для реализации вектор-функции $f_1(x, y) = \sin(x) \cos(y)$, $f_2(x, y) = y\sqrt{1+x^2}$. Использовать перегрузку операторов.

Задача 18. Написать программу с классом для реализации вектор-функции $\vec{f}(\vec{x}) = A\vec{x}$, где векторы \vec{f} и \vec{x} имеют размер n (целочисленная константа), а матрица A ранга n заполняется случайными целыми числами в диапазоне от 0 до 9 включительно. Использовать перегрузку операторов.

Задача 19. Написать программу с классом для реализации вектор-функции $\vec{f}(x) = A\vec{z}$, где векторы \vec{f} и \vec{z} имеют размер n (целочисленная константа), вектор-столбик $\vec{z} = (1, x, x^2, \dots, x^{n-1})^T$ («T» означает транспо-

нирование), а элементы a_{ij} матрицы A ранга n – случайные числа в диапазоне значений от 0 до 9 включительно ($i, j = 0, 1, \dots, n - 1$). Использовать перегрузку операторов.

Задача 20. Написать программу с классом для реализации векторов. При помощи перегрузки операторов реализовать процесс вычисления линейных комбинаций (сумма или разность векторов, умноженных на числовые коэффициенты). Предусмотреть возможность как умножения вектора на число, так и числа на вектор.

Глава 11

Наследование и виртуальные функции

Одним из фундаментальных механизмов, обеспечивающих уникальную гибкость и продуктивность программ, написанных на С++ (в рамках ООП), является **наследование**. Наследование позволяет одним объектам получать свойства других объектов. Эта глава посвящена наследованию и технологиям, непосредственно базирующимся на наследовании.

Наследование классов и типы наследования

*Желание нравиться своему веку
часто бывает поводом к тому,
чтобы не нравиться потомству.*

П. Мариво

Представим себе следующую ситуацию. В программе создан класс, у него есть определенные свойства (методы и поля), этот класс используется для создания объектов. В силу каких-то причин необходимо усовершенствовать, создать более сложный класс. Разумеется, можно взять имеющийся программный код как исходный и внести в него необходимые изменения. Но это не очень удобно в силу нескольких причин. Во-первых, редактирование старого проверенного кода не всегда просто сделать, поскольку изменения необходимо вносить так, чтобы не пострадали уже имеющиеся рабочие программы. Во-вторых, такой подход крайне неэффективен и непродуктивен и имеет ограниченную применимость. Необходимо постоянно контролировать вносимые изменения, что занимает время и стоит значительных усилий. Большие проблемы возникают, если на основе существующего кода необходимо создать несколько разных модификаций: в этом случае редактировать имеющийся код – совсем плохой вариант. Сложность программ растет, совместимость нарушается.

Разумеется, все эти неприятности не имело смысла перечислять, если бы в С++ не было эффективного механизма по их разрешению. Этот механизм – наследование.

В С++ классы могут наследовать друг друга. Это означает, что класс-наследник (в С++ он называется производным классом) получает свойства

своего родительского класса (родительский класс в C++ называется **базовым**). Кроме этого, производный класс может добавлять (обычно так и происходит) к свойствам, полученным от базового класса, свои собственные. Более того, производный класс может быть базовым по отношению к другому классу – все как в жизни. Сначала остановимся на наиболее простой ситуации, когда один класс наследует другой.

При наследовании классов производный класс фактически создается не на пустом месте, как обычно, а на основе базового класса, получая в наследство от своего родителя поля и методы. Тем не менее из того, что производный класс наследует свойства базового класса, еще не следует, что он наследует *все* свойства! Более точно, какие поля и методы наследуются, а какие нет, определяется доступностью соответствующих членов базового класса и механизмом наследования. Ранее уже отмечалось, что члены класса могут быть открытыми или закрытыми. Для объявления открытых членов класса используют идентификатор `public`. Закрытые члены класса объявляются с идентификатором доступа `private` (или без идентификатора доступа). Кроме того, члены класса могут объявляться с идентификатором доступа `protected`. В этом случае член класса называется защищенным. Такой член класса фактически является закрытым, а от `private`-члена он отличается способом наследования.

Ключевые слова `public`, `private` и `protected` используются не только для определения доступа к членам класса, но и являются индикаторами механизма наследования. Таким образом, в зависимости от типа наследования (`public`, `private` и `protected`) и доступности члена (`public`, `private` и `protected`) получаем разный результат. Например, `public`-член базового класса наследуется производным классом при любом механизме наследования, но в производном классе унаследованный член может иметь разный уровень доступа. При `public`-наследовании он останется `public`-членом в производном классе. При `private`-наследовании `public`-член базового класса становится `private`-членом производного класса. Несложно догадаться, что всего существует 9 возможных вариантов наследования. Все они перечислены в таблице 11.1.

Таблица 11.1. Типы наследования

Механизм наследования vs. тип члена	<code>public</code> -наследование	<code>private</code> -наследование	<code>protected</code> -наследование
<code>public</code> -член	<code>public</code>	<code>private</code>	<code>protected</code>
<code>private</code> -член	не наследуется	не наследуется	не наследуется
<code>protected</code> -член	<code>protected</code>	<code>private</code>	<code>protected</code>

Информацию о наследовании, представленную в виде таблицы 11.1, можно сформулировать в виде нескольких правил, которые приведем в порядке убывания важности (т.е. сначала применяется первое правило, затем второе и т.д.):

1. `private`-члены базового класса не наследуются;
2. при `public`-наследовании уровень доступа члена класса не меняется (если этот член наследуется);
3. при `private`-наследовании наследуемые члены становятся `private`-членами производного класса;
4. при `protected`-наследовании наследуемые члены становятся `protected`-членами производного класса.

Случай, когда член базового класса в производном классе не наследуется, потребует в дальнейшем некоторых пояснений. Пока же условимся под термином «не наследуется» подразумевать ситуацию, когда в производном классе соответствующий член отсутствует.

После того, как выяснены основные режимы наследования, разумно перейти к созданию производных классов на основе базовых. В частности, для того, чтобы на основе базового класса создать производный класс, необходимо при описании производного класса в прототипе после имени класса указать имя базового класса и механизм наследования. Синтаксис объявления производного класса в наиболее простом варианте имеет вид:

```
class производный_класс: тип_наследования базовый_класс{
    //программный код производного класса
}список_объектов;
```

Например, если класс В (производный класс) создается на основе класса А (базовый класс) через открытое наследование (`public`-наследование), соответствующий программный код объявления базового и производного классов будет следующим:

```
class A{
    //программный код класса А
}список_объектов;
//....
class B: public A{
    //программный код класса В
}список_объектов;
```

В этом случае все незакрытые члены класса А становятся членами класса В с таким же уровнем доступа. Пример использования наследования приведен в листинге 11.1.

Листинг 11.1. Пример простого наследования

```

#include <iostream>
using namespace std;
//Базовый класс:
class A{
    private:
        int x;
    public:
        int y;
};
//Производный класс:
class B: public A{
    public:
        int z;
        void show(){
            cout<<"y = "<<y<<endl;
            cout<<"z = "<<z<<endl;}
};
int main(){
    B obj;
    obj.y=1;
    obj.z=2;
    obj.show();
    return 0;}

```

В результате выполнения этой программы получим

```

y = 1
z = 2

```

В базовом классе А объявлено два поля типа `int`: поле `x` закрытое (`private`-член) и поле `y` открытое (`public`-член). Программный код класса В формально содержит описание всего одного `public`-поля `z` типа `int` и открытого `void`-метода `show()`. На то, что класс В является производным от класса А, указывает прототип описания класса В. Именно благодаря этому при описании метода `show()` допустимой является ссылка на поле `y` (хотя формально в классе В это поле не описано, оно описано в классе А), причем это поле объекта класса В. Объекты класса В получают это поле как наследство от класса А. К сожалению (или к счастью), этого нельзя сказать о поле `x`: поскольку поле в классе А объявлено как закрытое, оно не наследуется. Ссылка на это поле в классе В была бы некорректной. Кстати, стоит отметить, что в приведенном коде поле `x` нужно только для того, чтобы продемонстрировать, как поля *не наследуются*. Дело в том, что в классе А не предусмотрен механизм инициализации и изменения этого поля. Чтобы сделать поле функциональным, необходимо расширить программный код.

В методе `main()` создается объект класса `B`. Полям `y` и `z` этого объекта присваиваются значения, после чего значения полей отображаются с помощью метода `show()`.

Внесем в программный код некоторые изменения. Во-первых, в описании класса `A` поле `x` объявим как `protected`. При этом с точки зрения класса `A` поле продолжает оставаться закрытым, поскольку вне этого класса к полю доступ получить нельзя. Но теперь поле `x` наследуется классом `B`. У объектов класса `B` теперь будет, кроме полей `y` и `z`, еще и поле `x`. Правда, как и в случае класса `A`, к полю `x` объектов класса `B` доступ извне невозможен. Поэтому для проверки функциональности этого поля добавим вспомогательный программный код для метода `set()`, устанавливающего значение поля `x` (этот метод поместим в класс `A`, а в классе `B` он появится через механизм наследования), и внесем небольшие изменения в метод `show()`. Результат всех изменений представлен в листинге 11.2.

Листинг 11.2. Наследование `private`-члена

```
#include <iostream>
using namespace std;
//Базовый класс:
class A{
    protected:
        int x;
    public:
        int y;
        void set(int i){x=i;}
};
//Производный класс:
class B: public A{
    public:
        int z;
        void show(){
            cout<<"x = "<<x<<endl;
            cout<<"y = "<<y<<endl;
            cout<<"z = "<<z<<endl;}
};
int main(){
    B obj;
    obj.set(0);
    obj.y=1;
    obj.z=2;
    obj.show();
    return 0;}
```

Для инициализации значения поля `x` объекта `obj` класса `B` в главном методе программы вызывается метод `set()`, который наследуется из класса `A`.

Как отмечалось, напрямую обратиться к полю `x` в данном случае возможности нет – хотя поле и наследуется классом `B`, оно является защищенным (т.е. закрытым). Тем не менее, внутри класса `B` обращение к полю является корректным, например, как при описании метода `show()`. Результат выполнения программы имеет вид:

```
x = 0
y = 1
z = 2
```

Если теперь изменить механизм наследования, сделав, например, его `private`-наследованием, класс `B` будет иметь те же поля и методы, что и ранее, но теперь поля `x` и `y`, а также метод `set()` будут закрытыми. К этим членам класса нельзя будет обратиться вне класса. Возможны разные способы решения проблемы. Во-первых, можно предусмотреть возможность обращения к соответствующим полям, создав новые методы в производном классе. Такое задание оставляем для реализации читателю – ничего сложного или существенно нового в этом нет. Во-вторых, целесообразно переопределить метод `set()` так, чтобы в классе `B` этим методом устанавливались значения закрытых полей. Для этого необходимо объявить в классе `A` метод `set()` как виртуальный, а затем переопределить при наследовании. О виртуальных методах речь пойдет позже. Наконец, можно восстановить уровень доступа унаследованного члена класса. Так и поступим в данном случае.

Идея проста: если при наследовании члена класса уровень доступа к этому классу изменяется, в производном классе можно восстановить уровень доступа, который был у соответствующего члена в базовом классе. Фактически речь идет о том, чтобы при наследовании членов класса для некоторых членов сделать исключение. Делается такое исключение просто: в производном классе декларируется соответствующий член класса, перед которым, через оператор расширения контекста (двойное двоеточие `::`), указывается имя базового класса. Пример приведен в листинге 11.3.

Листинг 11.3. Восстановление уровня доступа при `private`-наследовании

```
#include <iostream>
using namespace std;
//Базовый класс:
class A{
    protected:
        int x;
    public:
        int y;
        void set(int i){x=i;}
};
```

```
//Производный класс (private-наследование):
class B: private A{
    public:
    int z;
    //Восстановление уровня доступа:
    A::y;
    A::set;
    void show(){
        cout<<"x = "<<x<<endl;
        cout<<"y = "<<y<<endl;
        cout<<"z = "<<z<<endl;}
};
int main(){
    B obj;
    obj.set(0);
    obj.y=1;
    obj.z=2;
    obj.show();
    return 0;}
```

Чтобы восстановить уровень доступа к полю `y`, использована инструкция `A::y`, а уровень доступа к методу `set()` восстанавливается командой `A::set`. Обращаем внимание читателя, что в этом случае круглые скобки для метода не указываются – только название метода.

Результат выполнения программы по сравнению с предыдущим случаем не меняется. Тем не менее увлекаться процедурой восстановления уровней доступа к наследуемым членам не следует. Во-первых, данная процедура не вписывается в концепцию инкапсуляции – одного из основополагающих принципов объектно-ориентированного программирования. Во-вторых, не исключено, что описанный выше механизм будет в будущем упразднен и станет недопустимым в стандарте языка C++. Благо, в C++ существует достаточно много гибких и эффективных способов для создания программ сложной структуры.

Переопределение методов и виртуальные функции

Прежде чем перейти непосредственно к обсуждению виртуальных функций и переопределению методов, остановимся кратко на идеях **полиморфизма**, широко используемых в объектно-ориентированном программировании. Ситуацию проиллюстрируем на достаточно простом примере. Представим, что имеется базовый класс `A`, у которого есть поле `x`. Изменять значение

этого поля можно с помощью метода `set()`, а отображается значение методом `get()`. Путем `public`-наследования создается класс В. Помимо унаследованного поля `x`, в классе содержится описание поля `y`. Класс также наследует методы `set()` и `get()`. Причем эти методы по умолчанию оперируют только с полем `x`, хотя вполне логично было бы, если бы в классе В метод `set()` определял значения обеих полей `x` и `y`, а метод `get()` отображал значения указанных полей. В этом случае был бы реализован единый интерфейс для определения и отображения значений полей объектов разных классов: и для класса А, и для класса В методом `set()` значения полей задаются, а методом `get()` значения полей отображаются. Такой подход очень просто реализуется, для чего достаточно в производном классе переопределить соответствующие методы. Фактически речь идет о том, что в производном классе унаследованный из базового класса метод определяется заново. Правда, на практике при переопределении метода в базовом классе он определяется как виртуальный. Признаком виртуального метода (т.е. такого метода, который переопределяется при наследовании) является ключевое слово `virtual`.

Прежде чем рассмотреть пример из листинга 11.4, сделаем одно очень важное замечание, касающееся разницы переопределения и перегрузки метода. При перегрузке, напомним, меняется прототип метода. При переопределении прототип метода остается неизменным.

Листинг 11.4. Переопределение и перегрузка методов при наследовании

```
#include <iostream>
using namespace std;
//Базовый класс:
class A{
    protected:
        int x;
    public:
        virtual void set(int i){x=i;}
        virtual void get(){
            cout<<"x = "<<x<<endl;}
};
//Производный класс:
class B: public A{
    private:
        int y;
    public:
        //Переопределение метода:
        void set(int i){
            x=i;
            y=i;}
};
```

```

//Перегрузка метода:
void set(int i,int j){
    x=i;
    y=j;}
//Переопределение метода:
void get(){
    cout<<"x = "<<x<<endl;
    cout<<"y = "<<y<<endl;}
};
int main(){
    A a;
    B b;
    a.set(1);
    a.get();
    b.set(2);
    b.get();
    b.set(3,4);
    b.get();
    return 0;}

```

При объявлении методов `set()` и `get()` в классе `A` использовано ключевое слово `virtual`. Благодаря этому в классе `B`, производном от класса `A`, методы `set()` и `get()` переопределяются. Причем если с методом `get()` все достаточно просто, то метод `set()` не только переопределяется, но еще и перегружается. Дело в том, что метод `get()` не содержит аргументов. Поэтому при изменении количества отображаемых полей в производном классе прототип метода не меняется. Отсюда и отсутствие какой бы то ни было интриги – прототипы метода в базовом и производном классах совпадают, изменяется только программный код основного тела метода. Впоследствии какой метод вызывать определяется по тому, из какого объекта вызывается метод. Например, если метод командой `a.get()` вызывается из объекта `a` класса `A`, то используется код метода `get()`, описанного в классе `A`. Если метод вызывается командой `b.get()`, то речь идет, очевидно, о методе `get()`, описанном в классе `B`, поскольку именно этому классу принадлежит объект `b`, из которого вызывается метод.

С методом `set()` дела обстоят сложнее. В классе `A` этот метод описан как такой, что имеет один аргумент – значение, которое присваивается полю `x`. В классе `B` значения нужно присваивать двум полям: полю `x` и полю `y`. Класс `B` содержит описание двух вариантов метода `set()` – с одним аргументом и с двумя аргументами. В случае, когда методу передается два аргумента, эти аргументы определяют значения полей `x` и `y` соответственно. При передаче методу одного аргумента поля `x` и `y` получают одинаковые значения. Прототип варианта метода `set()` с одним аргументом совпадает с прототипом метода `set()`, описанным в базовом классе. В этом случае

речь идет о переопределении метода. Перегрузка метода `set()` осуществляется путем определения в производном классе варианта метода `set()` с двумя аргументами. Программный код, используемый при вызове метода, определяется на основе того, из какого объекта вызывается метод и какой у метода прототип. В частности, в результате выполнения программы получим такой результат:

```
x = 1
x = 2
y = 2
x = 3
y = 4
```

Заметим, что для перегрузки метода в производном классе объявлять метод в базовом классе как виртуальный необходимости нет. Однако в этом случае необходимо учесть одну существенную тонкость. Поясним ее на примере. Допустим, что имеется класс `A`, у которого есть поле `n`, метод `f()`, результатом которого является удвоенное значение поля `n`, а также метод `show()`, в котором вызывается метод `f()` и полученный результат выводится на экран. На основе класса `A` путем наследования создается класс `B`, в котором метод `f()` переопределяется так, что вместо удвоенного значения поля `n` он возвращает в два раза меньшее значение. Сначала рассмотрим случай, когда метод `f()` объявлен в классе `A` как обычный, не виртуальный метод. Соответствующий программный код приведен в листинге 11.5.

Листинг 11.5. Переопределение не виртуального метода

```
#include <iostream>
using namespace std;
class A{
public:
    int n;
    int f(){return 2*n;}
    void show(){cout<<"show(): "<<f()<<endl;}
};
class B: public A{
public:
    int f(){return n/2;}
};
int main(){
    A a;
    a.n=3;
    B b;
    b.n=10;
    a.show();
    b.show();
}
```

```
cout<<"a.f() : "<<a.f()<<endl;
cout<<"b.f() : "<<b.f()<<endl;
return 0;}
```

В главном методе программы создаются два объекта (объект *a* класса *A* и объект *b* класса *B*). Полям *n* этих объектов присваиваются значения 3 и 10 соответственно. Далее для каждого из объектов вызывается метод `show()` и метод `f()`. Результат выполнения программы будет иметь вид:

```
show() : 6
show() : 20
a.f() : 6
b.f() : 5
```

В принципе, метод `show()` отображает значение, возвращаемое методом `f()`. В классе *B* метод `show()` наследуется из класса *A*, а вот метод `f()` в классе *B* переопределяется: в классе *A* этот метод возвращает число, вдвое большее поля *n*, а в классе *B* этот метод возвращает число, вдвое меньшее поля *n*. Результат выполнения программы свидетельствует о том, что метод `show()` класса *B* вызывает метод `f()`, но метод `f()` класса *A*! При этом если напрямую обратиться к методу `f()` класса *B*, будет вызвана корректная, переопределенная версия. С точки зрения объектно-ориентированной парадигмы такая ситуация является, мягко говоря, не очень разумной. Намного более логичным была бы ситуация, при которой метод `show()` класса *B* вызывал метод `f()` класса *B*. Решается эта проблема очень легко – метод `f()` объявляется как виртуальный, для чего в прототипе метода добавляется ключевое слово `virtual`. В листинге 11.6 приведен рассматривавшийся выше код с переопределением метода в качестве виртуального (изменения минимальны – добавлена одна инструкция).

Листинг 11.6. Переопределение виртуального метода

```
#include <iostream>
using namespace std;
class A{
public:
    int n;
    //Виртуальный метод:
    virtual int f(){return 2*n;}
    void show(){cout<<"show() : "<<f()<<endl;}
};
class B: public A{
public:
    int f(){return n/2;}
};
int main(){
```



```

A a;
a.n=3;
B b;
b.n=10;
a.show();
b.show();
cout<<"a.f(): "<<a.f()<<endl;
cout<<"b.f(): "<<b.f()<<endl;
return 0;}

```

Однако результат таких минимальных изменений существенный:

```

show(): 6
show(): 5
a.f(): 6
b.f(): 5

```

Обращаем внимание, что инструкция `virtual` используется только в классе `A` – виртуальность метода является наследуемой, поэтому в дальнейшем при переопределении метода этот атрибут указывать не нужно. Другими словами, если в базовом классе метод объявлен как виртуальный, он может быть переопределен не только в производном классе, но и в классах, созданных на основе производного класса. В этом случае говорят о многоуровневом наследовании.

Многоуровневое наследование

Никто не прибегает к аналогии, если можно ясно и четко выразить свою мысль.

А.И. Герцен

При многоуровневом наследовании, как отмечалось выше, производный класс, в свою очередь, является базовым для другого класса, который может быть базовым для следующего класса, и так далее. Образуется своеобразная иерархическая структура, в вершине которой находится базовый класс, на основе которого по цепочке последовательного наследования создаются новые классы. Проиллюстрируем сам процесс многократного наследования на примере.

В листинге 11.7 приведен пример многоуровневого наследования: класс `A` является базовым для класса `B`, а класс `B`, в свою очередь, является базовым для класса `C`, а класс `C` является базовым для класса `D`. При этом базовый класс `A` содержит виртуальные методы, которые переопределяются при наследовании.

Листинг 11.7. Многоуровневое наследование

```

#include <iostream>
using namespace std;
//Базовый класс A:
class A{
public:
    int n;
    virtual void set(int i){
        n=i;}
    virtual void get(){
        cout<<"n = "<<n<<endl;}
};
//Производный класс B:
class B: public A{
public:
    int m;
    void set(int i){
        n=i;
        m=i;}
};
//Производный класс C:
class C: public B{
public:
    void get(){
        cout<<"n-value is "<<n<<endl;
        cout<<"m-value is "<<m<<endl;}
};
//Производный класс D:
class D: public C{
public:
    int k;
};
int main(){
    A a;
    B b;
    C c;
    D d;
    a.set(1);
    b.set(2);
    c.set(3);
    d.set(4);
    a.get();
    b.get();
    c.get();
    d.get();
    return 0;}

```

В классе А всего одно целочисленное поле `n` и два метода: `set ()` с одним аргументом для присваивания значения полю `n` и `get ()` для отображения значения этого поля на экран. В классе В метод `set ()` переопределяется – методом присваиваются одинаковые значения полям `n` и `m` класса В. Метод `get ()` наследуется без изменений. В классе С переопределяется метод `get ()`, а в классе D – метод `set ()`. Результат выполнения программы имеет вид:

```
n = 1
n = 2
n-value is 3
m-value is 3
n-value is 4
m-value is 4
```

В главном методе программы создаются объекты всех четырех классов, и из каждого объекта вызывается по два метода: `set ()` и `get ()`. С объектом класса А все просто – вызываются те методы, что определены непосредственно в классе. При вызове методов из класса В метод вызывается «родной», а метод `get ()` наследуется из класса А. В классе С вызывается «родной» метод `get ()`, а метод `set ()` наследуется из класса В. Что касается класса D, то имеет место следующая ситуация: метод `set ()` наследуется по цепочке из класса В, а метод `get ()` наследуется из класса С.

Многократное наследование

*Цель оправдывает средства, кроме тех средств, которые под-
рывают саму цель.*

П.Л. Лавров

В C++ допустимой является ситуация, когда производный класс создается на основе сразу нескольких базовых классов. Такое наследование называется многократным. Такой симбиоз классов часто бывает исключительно продуктивным, хотя иногда и чреват неоднозначными ситуациями.

При многократном наследовании указывают, через запятую, базовые классы и механизм наследования для каждого из классов. Пример многократного наследования, когда производный класс создается на основе двух базовых классов, приведен в листинге 11.8.

Листинг 11.8. Многократное наследование

```
#include <iostream>
using namespace std;
//Первый базовый класс:
```

```

class A{
    public:
    int n;
    void shown() {
        cout<<"n = "<<n<<endl;}
};

//Второй базовый класс:
class B{
    public:
    int m;
    void showm() {
        cout<<"m = "<<m<<endl;}
};

//Производный класс на основе двух базовых:
class C: public A, public B{
    public:
    void show() {
        shown();
        showm();}
};

int main(){
    C obj;
    obj.n=3;
    obj.m=5;
    obj.show();
}

```

В программе объявляется три класса: два базовых и один производный от них. В классе А имеется целочисленное поле `n` и метод `shown()` для отображения значения этого поля. Аналогично в классе В объявлено целочисленное поле `m` и метод `showm()`, отображающий значение поля. Класс С создается на основе класса А и класса В, в силу чего наследуются все поля и методы этого класса. Кроме того, в классе С объявляется метод `show()` для отображения значений полей класса, причем реализован метод `show()` через вызов унаследованных методов `shown()` и `showm()`. Результат выполнения программы имеет вид

```

n = 3
m = 5

```

При многократном наследовании могут возникать проблемы с неоднозначностью наследования членов базовых классов. Например, предположим, что в предыдущем примере наследуемые поля классов А и В имеют одинаковые названия, скажем, `n`. В этом случае возникает неоднозначность: производный класс наследует поля с одинаковыми названиями. При обращении к таким полям необходимо явно указывать, из какого класса они наследо-

ваны. Например, если `obj` является объектом класса `C`, который наследует классы `A` и `B` с полями `n` в каждом, то обращение к полю, унаследованному от класса `A`, выглядит как `obj.A::n`. К полю, унаследованному от класса `B`, можно обратиться с помощью инструкции `obj.B::n`. Ситуация бывает и более замысловатой. Так, если исходный класс `A` является базовым для классов `B` и `C`, на основе которых создается класс `D`, то, во-первых, однозначно возникает проблема дублирования полей, а во-вторых, такое дублирование не так очевидно, как в предыдущем случае. Из такой ситуации выход может состоять, как и ранее, в явном указании класса, из которого наследуется поле, либо через виртуальное наследование базового класса `A`. При виртуальном наследовании совпадающие поля не дублируются, а реализуется виртуальное наследование с помощью ключевого слова `virtual`, указанного перед индикатором типа наследования. Пример приведен в листинге 11.9.

Листинг 11.9. Виртуальное наследование

```
#include <iostream>
using namespace std;
//Базовый класс:
class A{
    public:
        int n;
};
//Первый производный класс (виртуальное наследование):
class B: virtual public A{
    public:
        void show(){
            cout<<"n = "<<n<<endl;}
};
//Второй производный класс (виртуальное наследование):
class C: virtual public A{
    public:
        void get(){
            cout<<"n = "<<n<<endl;}
};
//Производный класс на основе двух базовых:
class D: public B, public C{
    public:
        void showAll(){
            show();
            get();}
};
int main(){
```

```
D obj;
obj.n=10;
obj.showAll();
return 0;}
```

У класса А всего одно целочисленное поле `n`, которое наследуется классами В и С (режим виртуального наследования). В классе В описан метод `show()` для значения унаследованного поля, а в классе С для тех же целей создан метод `get()`. В классе D, созданном на основе классов В и С, создается метод `showAll()`, в котором вызываются методы `show()` и `get()`. Поскольку базовыми (для класса D) классами В и С наследование класса А выполняется виртуально, дублирования поля `n` в классе D нет, и в результате выполнения программы получаем:

```
n = 10
n = 10
```

Тем не менее, желательно избегать ситуаций, при которых процесс наследования может интерпретироваться неоднозначно.

Конструкторы и деструкторы при наследовании

Важным и принципиальным является вопрос использования конструкторов и деструкторов при наследовании. Действительно, если в базовом классе имеется конструктор с аргументами, то в производном классе как минимум должен быть предусмотрен механизм передачи аргументов этому конструктору. Такой механизм, разумеется, существует. С точки зрения синтаксиса языка C++ сводится он к тому, что при определении конструктора производного класса после имени конструктора указывается имя конструктора базового класса с указанием в круглых скобках аргументов, передаваемых этому конструктору. Если базовых классов несколько, для каждого из них через запятую перечисляются конструкции вида `имя_конструктора (аргументы)` для передачи аргументов конструктору соответствующего базового класса. Синтаксис объявления конструктора производного класса имеет вид:

```
имя_конструктора (аргументы) : имя_конструктора1 (аргументы) ,
имя_конструктора (аргументы) , ... {
//код конструктора производного класса
}
```

Пример определения конструктора при наследовании приведен в листинге 11.10.

Листинг 11.10. Конструктор и наследование

```

#include <iostream>
using namespace std;
class Base{
public:
    int n;
    Base(int i){n=i;}
};
class Derivative:public Base{
public:
    int m;
    Derivative(int i,int j):Base(i){m=j;}
};
int main(){
    Base obj1(1);
    Derivative obj2(2,3);
    cout<<"obj1.n = "<<obj1.n<<endl;
    cout<<"obj2.n = "<<obj2.n<<endl;
    cout<<"obj2.m = "<<obj2.m<<endl;
    return 0;
}

```

В базовом классе `Base` объявлен один конструктор, предусматривающий передачу аргумента. Этим конструктором инициализируется значение целочисленного поля `n` класса. При создании производного класса `Derivative` необходимо предусмотреть передачу аргумента конструктору базового класса. Причина в способе создания объектов производных классов. Дело в том, что при создании объектов производного класса сначала вызывается конструктор базового класса, а уже затем конструктор класса производного. Если базовых классов несколько, то их конструкторы вызываются в порядке наследования классов (очередность в списке наследуемых базовых классов слева направо).

В конструкторе производного класса `Derivative` предусмотрена передача двух аргументов: один передается конструктору базового класса, а значение второго аргумента используется для инициализации целочисленного поля `m`, описанного в производном классе.

В главном методе программы создается объект базового класса и объект производного класса. В первом случае конструктору передается один аргумент, во втором – два. В результате выполнения программы получаем

```

obj1.n = 1
obj2.n = 2
obj2.m = 3

```

Пример определения конструктора производного класса при многократном наследовании приведен в листинге 11.11.

Листинг 11.11. Конструктор и многократное наследование

```

#include <iostream>
using namespace std;
class Base1{
public:
    int l;
    Base1(int i){
        cout<<"Base1-object created!\n";
        l=i;
        cout<<"l = "<<l<<endl;
    }
    ~Base1(){cout<<"Base1-object deleted!\n";};
};
class Base2{
public:
    int n;
    Base2(int i){
        cout<<"Base2-object created!\n";
        n=i;
        cout<<"n = "<<n<<endl;
    }
    ~Base2(){cout<<"Base2-object deleted!\n";};
};
class Derivative:public Base1,public Base2{
public:
    int m;
    Derivative(int i,int j,int k):Base1(i),Base2(j){
        cout<<"Derivative-object created!\n";
        m=k;
        cout<<"m = "<<m<<endl;
    }
    ~Derivative(){
        cout<<"Derivative-object deleted:\n";
        cout<<"l = "<<l<<endl;
        cout<<"n = "<<n<<endl;
        cout<<"m = "<<m<<endl;
    }
};
int main(){
    Derivative obj2(1,2,3);
    return 0;}

```

Этот же пример иллюстрирует вызов деструкторов. Правило вызова деструкторов состоит в том, что деструкторы вызываются в обратном порядке, если сравнивать с последовательностью вызова конструкторов. Первым вызывается деструктор производного класса, а после этого деструкторы базовых классов (очередность вызова деструкторов – справа налево в списке наследования базовых классов). Для удобства при вызове конструкторов и деструкторов базовых и производного классов выводятся соответствующие сообщения.

Главный метод программы состоит всего из одной команды создания объекта производного класса. В результате выполнения программы получаем:


```

Base1-object created!
l = 1
Base2-object created!
n = 2
Derivative-object created!
m = 3
Derivative-object deleted:
l = 1
n = 2
m = 3
Base2-object deleted!
Base1-object deleted!

```

Вся работа по выводу сообщений выполняется конструкторами и деструкторами, которых в данном случае по три (конструктор и деструктор для каждого из двух базовых классов и одного производного). Конструкторы вызываются в такой последовательности: конструктор класса `Base1`, конструктор класса `Base2` и затем конструктор класса `Derivative`. Деструкторы вызываются в обратной последовательности: деструктор класса `Derivative`, деструктор класса `Base2` и, наконец, деструктор класса `Base1`.

Чисто виртуальные методы и абстрактные классы

Все новое – это основательно забытое старое.

Соломон

Чисто виртуальной функцией называется такая виртуальная функция, которая не имеет определения в базовом классе. Чтобы определить чисто виртуальную функцию, после прототипа функции необходимо указать оператор присваивания (знак равенства) и ноль. Для чисто виртуальной функции блок с кодом функции не указывается. Синтаксис объявления чисто виртуальной функции имеет вид

```
virtual тип_результата имя_функции(аргументы)=0;
```

Класс, содержащий хотя бы одну чисто виртуальную функцию, называется абстрактным.

Совершенно очевидно, что в производном классе чисто виртуальная функция должна быть переопределена, поскольку из базового класса эта функция вызвана быть не может – там для нее просто нет программного кода. На первый взгляд такое положение дел может показаться не очень приемле-

мым, но это только на первый взгляд. Причин, по которым в программах используются чисто виртуальные функции, несколько. Главная связана с тем, что чисто виртуальные функции очень четко вписываются в концепцию объектно-ориентированного программирования. С их помощью создаются абстрактные классы, которые служат своеобразным шаблоном, каркасом, для создания более сложных, производных классов. Кроме того, нередко складывается ситуация, когда в силу специфики решаемой задачи виртуальная функция должна быть определена явно в каждом производном классе, создаваемом на основе базового. Если в базовом классе создать не просто виртуальную функцию, а чисто виртуальную функцию, это позволит избежать ошибок, связанных с отсутствием переопределения виртуальной функции в производном классе. В листинге 11.12 приведен пример использования чисто виртуальной функции.

Листинг 11.12. Чисто виртуальная функция

```
#include <iostream>
#include <cmath>
using namespace std;
const double pi=3.1415;
class Figure{
public:
double R;
Figure() {R=1;}
virtual double area()=0;
};
class Circle:public Figure{
public:
double area(){
return pi*R*R;}
};
class Square:public Figure{
public:
double area(){
return R*R;}
};
class Triangle:public Figure{
public:
double area(){
return sqrt(3)*R*R/4;}
};
int main(){
Circle A;
Square B;
Triangle C;
```

```
cout<<"Circle: "<<A.area()<<endl;
cout<<"Square: "<<B.area()<<endl;
cout<<"Triangle: "<<C.area()<<endl;
return 0;}
```

В программе создается абстрактный класс `Figure` с числовым полем `R` (линейный размер геометрической фигуры), конструктором, которым полю `R` присваивается единичное значение, и чисто виртуальным методом `area()` для вычисления площади фигуры. На основе этого абстрактного класса создается три производных класса, в каждом из которых переопределяется метод `area()`. В классе `Circle` метод `area()` переопределяется для вычисления площади круга, в классе `Square` метод `area()` переопределяется для вычисления площади квадрата, а в классе `Triangle` методом `area()` вычисляется площадь равностороннего треугольника. Результат выполнения программы имеет вид:

```
Circle: 3.1415
Square: 1
Triangle: 0.433013
```

В данном случае наследование выдержано в стиле объектно-ориентированной парадигмы. Базовый (абстрактный) класс определяет структуру всех производных классов (каждый класс должен иметь поле, определяющее линейные размеры фигуры, и метод для вычисления площади фигуры). Конкретная реализация способа вычисления площади фигуры определяется в каждом производном классе по-своему.

Существующие несуществующие члены класса

*Больше всего мы гордимся тем,
чего у нас нет.*

Акутагава Рюноске

Выше неоднократно упоминалось, что при наследовании закрытые члены класса не наследуются. Однако более точным в данном случае был бы термин «теряют доступность». Во всяком случае, необходимо четко представлять природу и механизм такого «ненаследования». Дело в том, что хотя закрытые члены и не наследуются, с технической точки зрения они существуют, и под них отводится место в памяти. Поясним ситуацию на примере.

Предположим, что в базовом классе `A` объявлено закрытое поле `int m`, конструктор с аргументом для инициализации поля `m`, а также метод (открытый) `getm()` для отображения значения этого поля.

В производном классе В доопределяется еще одно поле `int n`, конструктор с двумя аргументами (первый передается конструктору базового класса А, вторым инициализируется значение открытого поля `n` класса В), а также метод `getnm()` для отображения значения полей: в этом методе вызывается унаследованный метод `getm()` отображения закрытого поля `m` базового класса А и команда отображения значения открытого поля `n` производного класса В. Соответствующий программный код приведен в листинге 11.13.

Листинг 11.13. Ненаследуемые члены класса

```
#include <iostream>
using namespace std;
class A{
    int m;
public:
    A(int i){m=i;}
    void getm(){
        cout<<"m = "<<m<<endl;}
};
class B:public A{
public:
    int n;
    B(int i,int j):A(i){n=j;}
    void getnm(){
        getm();
        cout<<"n = "<<n<<endl;}
};
int main(){
    B obj(10,20);
    obj.getnm();
    return 0;}
```

В главном методе программы создается объект производного класса с передачей конструктору двух аргументов (10 и 20), после чего из объекта вызывается метод отображения значений полей `getnm()`. Как ни странно это может показаться, но результат выполнения программы имеет вид:

```
m = 10
n = 20
```

Причина происходящего кроется в способе создания объектов производного класса. В этом случае сначала, как уже отмечалось, вызывается конструктор базового класса. Конструктором базового класса выделяется место под все члены, прописанные в базовом классе, в том числе и закрытые (в данном случае это поле `m`). Причем конструктор базового класса вызывается с тем аргументом, что передан через конструктор производного класса. Хотя

поле m закрытое, конструктор имеет к нему доступ (это доступ в пределах класса) и в соответствующее место в памяти заносится нужное значение. Затем вызывается конструктор производного класса. Этим конструктором выделяется место для дополнительных членов, описанных непосредственно в производном классе, а также заполняется значение поля n . Таким образом, после создания объекта производного класса технически закрытое поле m существует (имеется в виду область памяти, выделенная для такого поля), но прямого доступа к нему в производном классе нет. Однако к этому полю есть не прямой доступ: в методе `getnm()` вызывается унаследованный открытый метод `getm()`, у которого, в свою очередь, есть доступ к области памяти, предназначенной для хранения значения поля m .

Примеры решения задач

Далее приводятся примеры решения задач, в которых используется механизм наследования.

■ Комплексные числа

Как известно, любое комплексное число $z = x + iy$ может быть представлено и в тригонометрической форме $z = r \exp(i\varphi)$, где r – модуль комплексного числа, а φ – аргумент. Модуль r и аргумент φ связаны с действительной $\operatorname{Re}(z) = x$ и мнимой $\operatorname{Im}(z) = y$ частями комплексного числа соотношениями $x = r \cos(\varphi)$ и $y = r \sin(\varphi)$, что дает в свою очередь $r = \sqrt{x^2 + y^2}$ и $\operatorname{tg}(\varphi) = y/x$. В некоторых случаях удобно использовать алгебраическую форму числа, а в некоторых – тригонометрическую. Здесь создадим класс для реализации алгебраической формы комплексного числа. После этого расширим класс, путем наследования, так, чтобы для комплексного числа можно было использовать и тригонометрическое представление.

Поскольку пары значений действительная-мнимая части и модуль-аргумент не являются независимыми, необходимо предусмотреть механизм их синхронного изменения. Для этого соответствующие поля делаем защищенными. Доступ к этим полям для считывания и изменения реализуется с помощью специальных методов. Программный код, в котором реализован данный подход, приведен в листинге 11.14.

Листинг 11.14. Комплексные числа

```
#include <iostream>
#include <cmath>
using namespace std;
```

```

//Базовый класс:
class ComplAlg{
protected:
//Действительная и мнимая часть:
double x,y;
public:
//Заполнение полей класса:
void set(double x,double y){
    this->x=x;
    this->y=y;}
//Отображение комплексного числа:
virtual void show(){
    cout<<"alg: z=";
    if(y==0){
        cout<<x<<endl;
        return;}
    if(x!=0) cout<<x;
    if(x!=0&& y>0) cout<<"+";
    if(y!=1&& y!=-1) cout<<y;
    if(y== -1) cout<<"-";
    cout<<"i\n";}
//Конструктор (без аргументов):
ComplAlg(){
    x=0;
    y=0;}
//Конструктор (с аргументами):
ComplAlg(double x,double y){
    this->x=x;
    this->y=y;}
}z1,z2(2,-1);
//Производный класс:
class Compl:public ComplAlg{
protected:
//Модуль и аргумент:
double r,phi;
public:
//Заполнение полей класса (переопределение метода):
void set(double x,double y){
    ComplAlg::set(x,y);
    r=sqrt(x*x+y*y);
    phi=atan2(y,x);}
//Отображение числа (переопределение метода):
void show(){
    ComplAlg::show();
    cout<<"trig: z=";
    if(phi==0){
        cout<<r<<"\n";

```

```

        return;}
    if(r!=1) cout<<r;
    cout<<"exp(";
    if(phi!=1&&phi!=-1) cout<<phi;
    if(phi== -1) cout<<"-";
    cout<<"i)\n";}
//Конструктор без аргументов:
Compl():ComplAlg(){
    r=0;
    phi=0;}
//Конструктор с аргументами:
Compl(double x,double y):ComplAlg(x,y){
    r=sqrt(x*x+y*y);
    phi=atan2(y,x);}
}z3,z4(cos(1),-sin(1));
int main(){
    z1.show();
    z1.set(3,4);
    z1.show();
    z2.show();
    z3.set(-1,0);
    z3.show();
    z4.show();
    return 0;}

```

В базовом классе `ComplAlg` объявлены защищенные поля `x` и `y` (действительная и мнимая части). Для работы с этими полями описываются виртуальные методы `set()` и `show()`. У метода `set()` два аргумента, которые определяют действительную и мнимую части комплексного числа. Метод `show()` используется для отображения комплексного числа. Некоторая замысловатость кода метода объясняется теми правилами, которые применялись для отображения комплексного числа в алгебраической форме. Кратко они сводятся к тому, чтобы форма представления числа внешне совпадала со стандартной математической формой отображения комплексных чисел. Основные положения, реализованные в методе `show()`, следующие:

1. Если число действительное (мнимая часть равна нулю), нулевая мнимая часть не отображается.
2. Нулевая действительная часть не отображается.
3. Если мнимая часть по модулю равна единице, эта единица не отображается (но отображается знак – плюс или минус).

Для того чтобы легче было различать, о какой форме представления комплексного числа идет речь, в начале строки вывода отображается метка `alg`.

В классе описаны два конструктора: без аргументов и с двумя аргументами. Конструктором без аргументов создается нулевое комплексное число, а конструктор с двумя аргументами позволяет при создании задать действительную и мнимую части.

Производный класс `Comp1` создается на основе базового класса `Comp1Alg`. Помимо наследуемых полей, в классе добавляется еще два защищенных поля: модуль и аргумент комплексного числа. В этом классе также переопределяются методы `set()` и `show()`. Обращаем внимание, что в каждом из этих методов вызывается старая версия метода из базового класса: в этом случае перед именем метода, через оператор расширения контекста, указывается имя базового класса. Метод `set()` расширяется так, чтобы вместе с изменением полей `x` и `y` изменялись соответствующим образом поля `r` и `phi`. Метод `show()` переопределяется так, что кроме алгебраической формы числа отображается и его тригонометрическая форма (перед тригонометрической формой указывается метка `trig`). При отображении тригонометрической формы числа также применяются некоторые правила:

1. Единичный модуль не отображается.
2. В случае нулевого аргумента комплексная экспонента не отображается.
3. Единичный аргумент в показателе экспоненты не отображается – только мнимая единица (и, если необходимо, знак).

Аналогичным образом определяются и конструкторы производного класса.

В главном методе программы в основном выполняются команды по изменению значений полей объектов и отображению значений на экран. В результате выполнения программы получаем:

```
alg:   z=0
alg:   z=3+4i
alg:   z=2-i
alg:   z=-1
trig:  z=exp(3.14159i)
alg:   z=0.540302-0.841471i
trig:  z=exp(-i)
```

Само собой разумеется, что для практического использования следует как минимум, кроме описанных методов, определить методы и перегрузить операторы для выполнения основных арифметических операций с комплексными числами. Аналогичные задачи решались в предыдущих главах книги.

■ Наследование операторных функций

На примере класса для работы с комплексными числами проиллюстрируем еще одну интересную особенность наследования – наследование перегруженных операторных функций. Рассмотрим упрощенные версии базового и производного классов, посредством которых реализуются комплексные числа. Обратимся к программному коду, представленному в листинге 11.15.

Листинг 11.15. Наследование операторных функций

```
#include <iostream>
using namespace std;
//Базовый класс:
class Compl1{
public:
//Действительная и мнимая часть:
double Re,Im;
//Перегрузка оператора сложения:
Compl1 operator+(Compl1 obj){
    Compl1 tmp;
    tmp.Re=Re+obj.Re;
    tmp.Im=Im+obj.Im;
    return tmp;}
//Конструктор с двумя аргументами:
Compl1(double x,double y){
    Re=x;
    Im=y;}
//Конструктор без аргументов:
Compl1(){
    Re=0;
    Im=0;}
    }a1(1,2),a2,a3(3,4);
//Производный класс:
class Compl2:public Compl1{
public:
//Перегрузка оператора умножения:
Compl2 operator*(Compl2 obj){
    Compl2 tmp;
    tmp.Re=Re*obj.Re-Im*obj.Im;
    tmp.Im=Re*obj.Im+Im*obj.Re;
    return tmp;}
//Конструктор с аргументами:
Compl2(double x,double y):Compl1(x,y){}
//Конструктор без аргументов:
Compl2():Compl1(){}
    }b1(10,20),b2(30,40),b3;
```

```

//Внешняя операторная функция:
Comp12 operator-(Comp11 x, Comp11 y) {
    Comp12 tmp;
    tmp.Re=x.Re-y.Re;
    tmp.Im=x.Im-y.Im;
    return tmp;}
//Внешний метод для отображения полей объекта:
void show(Comp11 obj){
    cout<<"Re: "<<obj.Re<<endl;
    cout<<"Im: "<<obj.Im<<endl;}
int main() {
    //Сумма объектов:
    a2=a1+b1;
    show(a2);
    a2=b1+a1;
    show(a2);
    a2=b1+b2;
    show(a2);
    //Произведение объектов:
    b3=b1*b2;
    show(b3);
    //Разность объектов:
    a2=a1-a3;
    show(a2);
    b3=b1-b2;
    show(b3);
    return 0;}

```

В базовом классе `Comp11` поля `Re` и `Im` (действительная и комплексная части) объявлены как открытые. Класс имеет конструктор без аргументов и конструктор с двумя аргументами. Однако главное внимание в данном случае следует уделить перегруженному оператору сложения: при сложении двух объектов класса `Comp11` получаем объект того же класса, а его поля равны сумме соответствующих полей исходных объектов. В этом смысле переопределение оператора достаточно стандартное. Интересно будет проследить возможности использования оператора сложения с учетом наличия производного класса.

В производном классе `Comp12` определяются конструкторы (без аргументов и с двумя аргументами, причем все действие этих конструкторов сводится к вызову соответствующего конструктора базового класса). Кроме конструкторов, в классе перегружается оператор умножения.

Представляет интерес также внешняя операторная функция, переопределяющая оператор вычитания. Сразу обращаем внимание читателя, что аргументами функции являются объекты класса `Comp11`, а результат – объект класса `Comp12`. Сделано это для того, чтобы с помощью данной функции

можно было вычислять разность не только объектов класса `Comp11`, но и объектов класса `Comp12`.

Внешняя функция `show()` используется для отображения значения полей объекта, указанного аргументом функции. В прототипе функции указано, что аргументом является объект класса `Comp11`.

В главном методе программы проверяется работа перегруженных операторов при работе с объектами базового и производного классов. В результате выполнения программы получаем:

```
Re: 11
Im: 22
Re: 11
Im: 22
Re: 40
Im: 60
Re: -500
Im: 1000
Re: -2
Im: -2
Re: -20
Im: -20
```

Для понимания принципов выполнения программного кода необходимо принять во внимание несколько важных обстоятельств.

Во-первых, операторные функции наследуются. Во-вторых, объект производного класса может присваиваться в качестве значения объекту базового класса. Этими двумя обстоятельствами объясняется возможность выполнять или не выполнять те или иные операции. Так, например, могут складываться объекты класса `Comp11` и `Comp12`. Результат присваивается объекту класса `Comp11` (команда `a2=a1+b1` в главном методе программы). Напомним, что операторная функция для оператора сложения описана в классе `Comp11`. При вычислении результата выражения `a1+b1` операторная функция вызывается из объекта `a1`, а ее формальным аргументом является объект `b1` класса `Comp12`. В прототипе операторной функции аргумент относится к классу `Comp11`. Однако поскольку объекты производного класса могут присваиваться объектам базового класса, аргументом операторной функции может быть и объект производного класса `Comp12`.

Несколько иная ситуация имеет место при выполнении команды `a2=b1+a1`, в которой к объекту класса `Comp12` прибавляется объект класса `Comp11`. В этом случае операторная функция для оператора сложения вызывается из объекта `b1` класса `Comp12`, что становится возможным благодаря наследованию соответствующей операторной функции. В силу тех же при-

чин легитимна и команда $a2=b1+b2$: операторный метод вызывается из объекта класса `Comp12` благодаря наследованию, а второй операнд является объектом класса `Comp12` на правах наследника базового класса.

Этого никак нельзя сказать о произведении объектов: соответствующая операторная функция объявлена в производном классе `Comp12`, вызывается из объекта этого класса, ее аргументом является объект класса `Comp12`, результатом является объект того же класса. Команда $b3=b1*b2$ иллюстрирует ситуацию.

Команды $a2=a1-a3$ и $b3=b1-b2$ иллюстрируют возможности внешней операторной функции для оператора вычитания. В качестве результата функцией возвращается объект класса `Comp12`, поэтому результат вычисления разности двух объектов можно записать не только в объект класса `Comp12`, но и в объект базового класса `Comp11`. Аргументами функции являются объекты класса `Comp11`, поэтому в качестве аргументов можно передавать и объекты производного класса `Comp12` (оба аргумента или один из них). По той же причине аргументом функции `show()` могут передаваться объекты обоих классов.

■ Преобразование Фурье

Процедура наследования может успешно применяться в самых разных ситуациях. Здесь рассмотрим пример программы, в которой создаются классы для выполнения преобразований Фурье произвольной функции на интервале от $-\pi$ до π . В частности, создается два базовых класса: в одном классе реализуется разложение по косинусам, а в другом – по синусам. Напомним, что в общем случае представлением Фурье для функции $f(x)$ на интервале $(-L, L)$ называется ряд

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos\left(\frac{\pi n x}{L}\right) + b_n \sin\left(\frac{\pi n x}{L}\right) \right), \text{ где коэффициенты разложения } a_n = \frac{1}{L} \int_{-L}^L f(x) \cos\left(\frac{\pi n x}{L}\right) dx \text{ и } b_n = \frac{1}{L} \int_{-L}^L f(x) \sin\left(\frac{\pi n x}{L}\right) dx.$$

Если разложение выполняется на интервале $(-\pi, \pi)$, то ряд Фурье имеет вид $f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(nx) + b_n \sin(nx))$ с коэффициентами

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx) dx \text{ и } b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx) dx. \text{ Если функ-}$$

ция $f(x)$ обладает определенной симметрией, ситуация упрощается. Для симметричной функции $f(-x) = f(x)$ имеет место косинус-разложение

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos\left(\frac{\pi n x}{L}\right), \quad \text{где} \quad b_n = \frac{2}{L} \int_0^L f(x) \sin\left(\frac{\pi n x}{L}\right) dx. \quad \text{Если}$$

данными формулами воспользоваться для представления в виде ряда для функции, не обладающей симметрией $f(-x) = f(x)$, получим, фактически, представление функции на интервале $(0, L)$ с симметричным отображением относительно оси ординат. Аналогично для антисимметричных функций

$$f(-x) = -f(x) \quad \text{имеет место синус-разложение} \quad f(x) = \sum_{n=1}^{\infty} b_n \sin\left(\frac{\pi n x}{L}\right) \text{ с}$$

коэффициентами $b_n = \frac{2}{L} \int_0^L f(x) \sin\left(\frac{\pi n x}{L}\right) dx$. Разложение по синусам

произвольной функции соответствует разложению этой функции в ряд на интервале $(0, L)$ с антисимметричным отображением относительно начала координат.

Для выполнения преобразования Фурье необходимо вычислить интеграл (для каждого преобразования свой). Функции `IntCos()` и `IntSin()` для вычисления интегралов от раскладываемой в ряд функции, умноженной на синус или косинус, описаны в каждом из двух базовых классов `FourierCos` и `FourierSin` как защищенные методы. Методы предполагают два способа вычисления коэффициентов ряда Фурье: если аргументом метода указано логическое значение `false` или аргумент не указан вообще, коэффициент косинус-разложения или синус-разложения вычисляется в предположении, что раскладываемая в ряд функция четная или нечетная соответственно (интеграл вычисляется от 0 до π). При аргументе `true` соответствующий интеграл вычисляется на интервале от $-\pi$ до π . Кроме аргумента логического типа, методам передается целочисленный аргумент, определяющий индекс вычисляемого коэффициента.

Что касается способа вычисления интеграла, то использован метод трапеций: весь диапазон интегрирования разбивается на равные интервалы, значения функции в узловых точках соединяются прямыми линиями, а интеграл вычисляется как сумма площадей полученных таким образом трапеций. Если интеграл S вычисляется на диапазоне от a до b от функции

$$f(x), \quad \text{то используется формула} \quad S \approx \frac{h}{2} \sum_{n=0}^{N-1} (f(a + nh) + f(a + (n+1)h)),$$

где $h = (b - a)/N$, N – количество интервалов, на которые разбит диапа-

зон интегрирования. Чем больше этих интервалов, тем точнее вычисляется интеграл.

В базовых классах `FourierCos` и `FourierSin` оператор `()` перегружается так, что если аргументом указано число типа `double`, то в качестве результата возвращается значение ряда Фурье для данной точки. Раскладываемая в ряд функция определяется полем класса `double (*f)(double)` (поле `f`), которое является указателем на функцию от действительного аргумента. Значение ряда вычисляется на основе массива `double a[n+1]` (косинус-преобразование) или `double b[n+1]` (синус-преобразование). Массивы заполняются при вызове метода `make()`. Поскольку в методе вызывается, в свою очередь, метод для вычисления интеграла `IntCos()` или `IntSin()`, методу `make()` может передаваться логический аргумент, с которым потом будет вызван метод `IntCos()` или `IntSin()`.

Если аргументом в круглых скобках после имени объекта класса `FourierCos` или `FourierSin` указано имя функции, то ссылка на эту функцию будет присвоена полю `f` соответствующего объекта.

Производный класс `Fourier` для выполнения полного преобразования Фурье создается на основе базовых классов `FourierCos` и `FourierSin`. В нем переопределяются все методы, наследуемые из базовых классов, причем в основном переопределение технически реализуется через вызов исходных версий методов базовых классов. Обращаем внимание читателя на то, что производный класс `Fourier` наследует из базовых классов `FourierCos` и `FourierSin` два поля с одинаковым названием `f`. Поэтому при обращении к этим полям в производном классе явно указывается, из какого базового класса они наследованы. При заполнении этим полям присваивается одно и то же значение. Читатель может самостоятельно предложить пути иного выхода из этой ситуации. Описанный выше программный код приведен в листинге 11.16.

Листинг 11.16. Ряд Фурье

```
#include <iostream>
#include <cmath>
using namespace std;
//Количество слагаемых ряда:
const int n=100;
const double pi=3.1415;
//Класс для косинус-преобразования:
class FourierCos{
protected:
    //Раскладываемая в ряд функция:
    double (*f)(double);
```

```

//Коэффициенты разложения:
double a[n+1];
//Интеграл для косинус-преобразования:
double IntCos(int k,bool state=false){
int N=10000,i;
    double s=0,h=(1+state)*pi/N,x=-pi*state;
    for(i=0;i<N;i++){
        s+=(f(x)*cos(x*k)+f(x+h)*cos((x+h)*k))/2;
        x+=h;}
    s*=h*(2-state);
    return s;}
public:
//Метод для вычисления коэффициентов ряда:
void make(bool state=false){
    int i;
    for(i=0;i<=n;i++){
        a[i]=IntCos(i,state)/pi;}
    a[0]/=2;}
//Перегрузка оператора ():
double operator()(double x){
    double s=0;
    for(int i=0;i<=n;i++)
        s+=a[i]*cos(x*i);
    return s;}
void operator()(double (*f)(double)){
    this->f=f;}
}F1;
//Класс для синус-преобразования:
class FourierSin{
protected:
//Раскладываемая в ряд функция:
double (*f)(double);
//Коэффициенты разложения:
double b[n+1];
//Интеграл для синус-преобразования:
double IntSin(int k,bool state=false){
    int N=10000,i;
    double s=0,h=(1+state)*pi/N,x=-pi*state;
    for(i=0;i<N;i++){
        s+=(f(x)*sin(x*k)+f(x+h)*sin((x+h)*k))/2;
        x+=h;}
    s*=h*(2-state);
    return s;}
public:
//Метод для вычисления коэффициентов ряда:
void make(bool state=false){

```

```

    int i;
    for (i=1; i<=n; i++) {
        b[i]=IntSin(i, state)/pi;
    }
    b[0]=0;
}
//Перегрузка оператора ():
double operator() (double x) {
    double s=0;
    for (int i=1; i<=n; i++)
        s+=b[i]*sin(x*i);
    return s;
}
void operator() (double (*f) (double)) {
    this->f=f;
}
}F2;
//Класс для преобразования Фурье:
class Fourier:public FourierCos,public FourierSin{
public:
    //Переопределение метода для вычисления коэффициентов ряда:
    void make(bool state=false) {
        FourierCos::make(true);
        FourierSin::make(true);
    }
    //Переопределение оператора ():
    double operator() (double x) {
        return FourierCos::operator() (x)+FourierSin::operator() (x);
    }
    void operator() (double (*f) (double)) {
        this->FourierCos::f=f;
        this->FourierSin::f=f;
    }
}F;
//Функции для преобразования Фурье:
double f1(double x){
    return x*x;
}
double f2(double x){
    return x;
}
double f(double x){
    return f1(x)+f2(x);
}
int main() {
    F1(f1);
    F1.make();
    F2(f2);
    F2.make();
    F(f);
    F.make();
    cout<<F1(1)<<endl;
    cout<<F2(2)<<endl;
    cout<<F(-3)<<endl;
    return 0;
}

```


В результате выполнения программы получим:

```
0.999924
2.00106
5.99547
```

Это результаты вычисления значения ряда Фурье для разных аргументов функций $f_1(x) = x^2$, $f_2(x) = x$ и $f(x) = f_1(x) + f_2(x) = x^2 + x$. Применялось соответственно косинус-преобразование, синус-преобразование и полное преобразование Фурье. Предлагаем читателю самостоятельно объяснить результат выполнения программы.

■ Произведение полиномов и ряд Тейлора

Рассмотрим задачу о вычислении произведения полиномов, которую будем решать с привлечением процедуры наследования. В качестве базового создаем класс `Polynom1` для реализации полиномов степени не выше, чем n (глобальная константа), в котором определяется поле-массив с коэффициентами полинома. Для класса перегружаются операторы `[]` и `()` для прямого доступа к коэффициентам полинома и вычисления значения полинома в точке соответственно. В принципе, для работы с полиномами можно определить ряд операций, таких как сложение, вычитание, умножение и деление на число, вычисление производной. Это те операции, которые позволяют представить результат их выполнения в виде объекта класса `Polynom1`. Другими словами, они не приводят к выходу за пределы этого класса. Если операндами являются полиномы степени не выше n , то результатом также является полином степени не выше n — чего нельзя сказать о произведении полиномов. Результатом произведения полиномов степени не выше n является полином степени не выше $2n$. Чтобы можно было вычислить произведение полиномов, на основе класса `Polynom1` создаем, путем наследования, класс `Polynom2` для реализации полиномов степени не выше $2n$. Программный код приведен в листинге 11.17.

Листинг 11.17. Произведение полиномов и ряд Тейлора

```
#include <iostream>
#include <cmath>
using namespace std;
//Степень базового полинома:
const int n=5;
//Базовый класс:
class Polynom1{
public:
    //Коэффициенты полинома:
    double a[n+1];
```

```

//Перегрузка оператора []:
virtual double &operator[](int k){
    return a[k];}
//Перегрузка оператора ():
virtual double operator()(double x){
    int i;
    double s=0;
    for(i=0;i<=n;i++)
        s+=a[i]*pow(x,i);
    return s;}
//Метод для отображения коэффициентов:
virtual void show(){
    int i;
    for(i=0;i<=n;i++)
        cout<<a[i]<<" ";
    cout<<endl;}
}P1,P2,P3;
//Производный класс:
class Polynom2:public Polynom1{
public:
    //"Дополнительные" коэффициенты:
    double b[n];
    //Переопределение оператора []:
    double &operator[](int k){
        if(k<=n) return a[k];
        else return b[k-n-1];}
    //Переопределение оператора ():
    double operator()(double x){
        int i;
        double s=a[0];
        for(i=1;i<=n;i++)
            s+=(a[i]*pow(x,i)+b[i-1]*pow(x,n+i));
        return s;}
    //Переопределение метода для отображения коэффициентов:
    void show(){
        int i;
        for(i=0;i<=n;i++)
            cout<<a[i]<<" ";
        for(i=0;i<n;i++)
            cout<<b[i]<<" ";
        cout<<endl;}
}Q;
//Внешняя функция для вычисления произведения полиномов:
Polynom2 operator*(Polynom1 obj1,Polynom1 obj2){
    Polynom2 tmp;
    int i,j;

```

```

    for(i=0;i<=n;i++){
        tmp[i]=0;
        for(j=0;j<=i;j++){
            tmp[i]=tmp[i]+obj1[j]*obj2[i-j];
        }
    }
    for(i=n+1;i<=2*n;i++){
        tmp[i]=0;
        for(j=i-n;j<=n;j++){
            tmp[i]=tmp[i]+obj1[j]*obj2[i-j];
        }
    }
    return tmp;}

int main(){
    int i;
    //Инициализация полиномов:
    P1[0]=0;
    P2[0]=1;
    for(i=1;i<=n;i++){
        P1[i]=(double)(2*(i%2)-1)/i;
        P2[i]=(i+1)%2;
    }
    //Произведение полиномов:
    Q=P1*P2;
    //Ряд Тейлора:
    P3=P1*P2;
    P1.show();
    P2.show();
    Q.show();
    P3.show();
    //Проверка значений:
    cout<<"P1: "<<P1(1)<<endl;
    cout<<"P2: "<<P2(1)<<endl;
    cout<<"P3: "<<P3(1)<<endl;
    cout<<"Q: "<<Q(1)<<endl;
    cout<<"P1*P2: "<<P1(1)*P2(1)<<endl;
    return 0;}

```

Что касается технической реализации процедуры наследования, то в данном случае для записи коэффициентов возле слагаемых с показателями аргумента, большими n , в производном классе определяется массив b . Путем переопределения операторной функции `[]` выполняется индексация объектов массива так, что наличие двух массивов вместо одного, как в базовом классе, никак не сказывается на способе обращения к коэффициентам полинома – после имени соответствующего объекта в квадратных скобках указывается индекс коэффициента (индекс совпадает с показателем степени аргумента возле соответствующего коэффициента).

Для удобства в программе оператор $()$ перегружается в базовом классе для вычисления значения полинома, а в производном классе этот оператор переопределяется с учетом расширенного набора коэффициентов. Таким образом, для вычисления значения полинома при заданном аргументе – для этого достаточно после имени соответствующего объекта указать в круглых скобках аргумент.

Непосредственно произведение полиномов может быть вычислено с помощью внешней операторной функции. У функции два аргумента класса `Polynom1`, а результатом является объект класса `Polynom2`. При вычислении полинома-произведения использовано следующее правило. Если полином $P_1(x) = \sum_{k=0}^n a_k x^k$ умножается на полином $P_2(x) = \sum_{k=0}^n b_k x^k$, в результате получим полином

$$Q(x) = P_1(x)P_2(x) = \sum_{i=0}^n \sum_{j=0}^{i-j} a_j b_{i-j} x^i + \sum_{i=n+1}^{2n} \sum_{j=i-n}^n a_j b_{i-j} x^i.$$

Именно с учетом данного соотношения заполняются элементы массивов объекта производного класса. При этом обращение к элементам массива осуществляется путем индексации объектов, возможной благодаря перегрузке и переопределению оператора `[]`.

Поскольку в качестве результата операторной функцией умножения возвращается объект класса `Polynom2`, который является производным от класса `Polynom1`, результат произведения двух полиномов можно записать (присвоить в качестве значения) в объект базового класса `Polynom1`. Поскольку в классе `Polynom1` предусмотрена возможность обработки только части коэффициентов массива (в массиве `double a` всего n элементов), то при присваивании результата объекту класса `Polynom1` коэффициенты старших слагаемых с показателями степени от $n+1$ до $2n$ теряются (теряются значение массива `double b` класса `Polynom2`). На первый взгляд это может показаться недостатком, но это не так. Данная особенность может с успехом применяться при вычислении ряда Тейлора для различных функций. Напомним, что рядом Тейлора для функции $f(x)$ в окрестности точки x_0 называется представление функции в виде бесконечной суммы $f(x) = \sum_{n=0}^{\infty} c_n (x - x_0)^n$, где коэффициенты c_n ряда определяются через производные $f^{(n)}(x_0)$ от исходной функции в точке разложения $c_n = \frac{f^{(n)}(x_0)}{n!}$. Нередко задача формулируется так: необходимо определить ряд Тейлора (определенное количество слагаемых ряда) для функции

$f(x) = f_1(x)f_2(x)$, если известны разложения для функций $f_1(x)$ и $f_2(x)$. В этом случае вычисляется произведение двух рядов, группируются слагаемые с одинаковыми показателями аргумента. При этом если требуется найти разложение до степени аргумента n включительно, два исходных ряда также ограничиваются слагаемыми соответствующего порядка, а в полиноме-результате все слагаемые со степенями, большими n , отбрасываются (оставлять их нет никакого смысла, поскольку они не являются коэффициентами ряда Тейлора – при их вычислении в исходных полиномах не учитываются слагаемые более высокого порядка, чем n). Проиллюстрируем это на конкретном примере (который, кстати, нашел свое отображение в приведенном выше программном коде).

Необходимо определить ряд Тейлора для функции $f(x) = \ln(1+x)/(1-x^2)$ в окрестности точки $x_0 = 0$ до слагаемых степени $n \leq 5$, если

$$\ln(1+x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots$$

$$\text{и } 1/(1-x^2) = \sum_{n=0}^{\infty} x^{2n} = 1 + x^2 + x^4 + x^6 + \dots$$

Результатом является выражение $f(x) \approx x - \frac{x^2}{2} + \frac{4x^3}{3} - \frac{3x^4}{4} + \frac{23x^5}{15}$

или то же самое $f(x) \approx x - 0.5x^2 + 1.333333x^3 + 0.75x^4 + 1.533333x^5$. Здесь формально множится два полинома степени $n = 5$, а результатом является полином степени $2n = 10$, причем почти половину из слагаемых полинома следует отбросить. В этом случае как нельзя кстати пригодится особенность присваивания объекта производного класса объекту базового класса.

В главном методе программы создаются полиномы для функций $\ln(1+x)$ и $1/(1-x^2)$, затем вычисляется их произведение. Приводится полный результат и результат с учетом отбрасывания старших слагаемых. Таким образом, при выполнении программы получаем следующее:

```
0 1 -0.5 0.333333 -0.25 0.2
1 0 1 0 1 0
0 1 -0.5 1.333333 -0.75 1.533333 -0.75 0.5333333 -0.25 0.2 0
0 1 -0.5 1.333333 -0.75 1.533333
P1: 0.783333
P2: 3
P3: 2.61667
Q: 2.35
P1*P2: 2.35
```

В конце приводится результат вычисления значений исходных и полученных в результате умножения полиномов для единичного аргумента.

Резюме

1. В C++ классы могут наследовать друг друга. Класс-наследник (производный класс) получает свойства своего родительского класса (базовый класс). Производный класс может добавлять к свойствам, полученным от базового класса, свои собственные.
2. Существуют различные механизмы наследования. Ключевые слова `public`, `private` и `protected` используются не только для определения доступа к членам класса, но и являются индикаторами механизма наследования. В зависимости от типа наследования (`public`, `private` и `protected`) и доступности члена (`public`, `private` и `protected`) получаем разный результат на предмет доступности того или иного члена класса.
3. Наследуемые методы можно переопределять. При описании переопределяемых в производных классах методов их объявляют как виртуальные. Для этого в начале прототипа метода указывается ключевое слово `virtual`.
4. Свойство виртуальности является наследуемым.
5. Производный класс может быть базовым для другого класса. В этом случае говорят о многоуровневом наследовании.
6. Производный класс может создаваться на основе одновременно нескольких классов. В этом случае говорят о многократном наследовании.
7. Последовательность вызова конструкторов при создании объектов производного класса следующая: сначала вызывается конструктор первого наследуемого класса, затем второго и т.д. Последним вызывается конструктор производного класса (в списке наследования базовых классов – слева направо).
8. Последовательность вызова деструкторов при удалении объекта производного класса обратная к последовательности вызова конструкторов: первым вызывается деструктор производного класса, затем последнего базового класса из списка наследования и т.д. до первого наследуемого класса (в списке наследования базовых классов – справа налево).
9. Чисто виртуальным методом называется такой виртуальный метод, который не определен в базовом классе. Класс, содержащий хотя бы один чисто виртуальный метод, называется абстрактным.

Контрольные вопросы

Глупые мысли бывают у всякого, только умный их не высказывает.

В. Буш

1. Что такое наследование и в чем оно состоит?
2. Какие существуют типы наследования?
3. Что такое виртуальные методы? Как происходит переопределение виртуальных методов?
4. Что такое многоуровневое наследование?
5. Что такое многократное наследование?
6. В чем особенность конструктора производного класса?
7. Какова последовательность вызова конструкторов при создании объекта производного класса?
8. Какова последовательность вызова деструкторов при удалении объекта производного класса?
9. Что такое чисто виртуальный метод?
10. Что такое абстрактный класс?

Задачи для самостоятельного решения

Образование – клад, труд – ключ к нему.

П. Буаст

В предлагаемых для самостоятельного решения задачах необходимо применить методы наследования классов. При этом большинство примеров носят учебный характер. Они направлены на закрепление основных подходов, используемых при реализации наследования, поэтому достаточно просты.

Задача 1. Написать программу с базовым классом для реализации комплексных чисел в алгебраической форме и основных операций с ними: сложения, вычитания, умножения и деления. Путем наследования создать производный класс, в котором определить операции вычисления модуля комплексного числа и комплексно сопряженного.

Задача 2. Написать программу с классом для реализации комплексного числа в тригонометрической форме и основных операций с ними: сложения, вычитания, умножения и деления. Путем наследования создать производный класс, в котором определить операции вычисления действительной и мнимой частей комплексного числа, а также комплексно сопряженного.

Задача 3. Написать программу с классом для реализации комплексных чисел в алгебраической форме и определить основные операции: сложение, вычитание, умножение и деление (в том числе и на действительные числа). Создать производный класс, в котором определить метод для вычисления

$$\text{комплексной экспоненты } \exp(z) = 1 + z + \frac{z^2}{2!} + \dots + \frac{z^n}{n!} + \dots$$

Задача 4. Написать программу с классом для реализации комплексных чисел в алгебраической форме и определить основные операции: сложение, вычитание, умножение и деление (в том числе и на действительные числа). Создать производный класс, в котором определить метод для вычисления

$$\text{комплексного косинуса } \cos(z) = 1 - \frac{z^2}{2!} + \frac{z^4}{4!} - \dots + \frac{(-1)^n z^{2n}}{(2n)!} + \dots$$

Задача 5. Написать программу с классом для реализации комплексных чисел в алгебраической форме и определить основные операции: сложение, вычитание, умножение и деление (в том числе и на действительные числа). Создать производный класс, в котором определить метод для вычисления

$$\text{комплексного синуса } \sin(z) = z - \frac{z^3}{3!} + \frac{z^5}{5!} - \dots + \frac{(-1)^n z^{2n+1}}{(2n+1)!} + \dots$$

Задача 6. Написать программу с классом для реализации квадратных матриц предопределенного размера. Определить в этом классе операции сложения, вычитания, произведения матриц, а также умножения и деления матриц на число. Создать производный класс, в котором реализовать процедуру вычисления

$$\text{матричной экспоненты } \exp(A) = E + A + \frac{A^2}{2!} + \dots + \frac{A^n}{n!} + \dots,$$

где A – матрица-аргумент, а E – единичная матрица.

Задача 7. Написать программу с классом для реализации квадратных матриц предопределенного размера. Определить в этом классе операции сложения, вычитания, произведения матриц, а также умножения и деления матриц на число. Создать производный класс, в котором реализовать процедуру вычисления матричного косинуса

$$\cos(A) = E - \frac{A^2}{2!} + \frac{A^4}{4!} - \dots + \frac{(-1)^n A^{2n}}{(2n)!} + \dots, \text{ где } A - \text{матрица-аргумент,}$$

а E – единичная матрица.

Задача 8. Написать программу с классом для реализации квадратных матриц предопределенного размера. Определить в этом классе операции сложения, вычитания, произведения матриц, а также умножения и деления матриц на число. Создать производный класс, в котором реализовать процедуру вычисления матричного косинуса $\sin(A) = A - \frac{A^3}{3!} + \frac{A^5}{5!} \dots + \frac{(-1)^n A^{2n+1}}{(2n+1)!} + \dots$,

где A – матрица-аргумент.

Задача 9. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность складывать и вычитать комплексные числа. Создать производный класс для реализации векторов с комплексными компонентами (вектор имеет три компонента, каждый компонент – комплексное число). Компоненты вектора являются элементами массива. Предусмотреть возможность индексирования объектов и сложения и вычитания векторов.

Задача 10. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность сложения, вычитания и умножения комплексных чисел. Создать производный класс для реализации векторов с комплексными компонентами (вектор имеет три компонента, каждый компонент – комплексное число). Компоненты вектора являются элементами массива. Предусмотреть возможность вычисления скалярного произведения векторов – результатом является комплексное число, равное сумме произведений соответствующих компонентов векторов.

Задача 11. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность сложения, вычитания и умножения комплексных чисел, а также вычисление комплексно сопряженного числа. Создать производный класс для реализации векторов с комплексными компонентами (вектор имеет три компонента, каждый компонент – комплексное число). Компоненты вектора являются элементами массива. Предусмотреть возможность вычисления скалярного произведения векторов, а также вычисления модуля вектора (произведение вектора на комплексно сопряженный вектор).

Задача 12. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность сложения, вычитания и умножения комплексных чисел. Создать производный класс для реализации векторов с комплексными компонентами (вектор имеет три компонента, каждый компонент – комплексное число). Компоненты вектора являются элементами массива. Предусмотреть возможность вычисления векторного произведения векторов – результатом является комплексный вектор.

Задача 13. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность складывать и вычитать комплексные числа. Создать производный класс для реализации квадратных матриц с комплексными элементами. Элементы матрицы заносятся в двумерный массив. Предусмотреть возможность индексирования объектов и сложения и вычитания матриц.

Задача 14. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность складывать, вычитать и умножать комплексные числа. Создать производный класс для реализации квадратных матриц с комплексными элементами. Элементы матрицы заносятся в двумерный массив. Предусмотреть возможность индексирования объектов и умножения матриц.

Задача 15. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность складывать, вычитать и умножать комплексные числа. Создать производный класс для реализации квадратных матриц с комплексными элементами. Элементы матрицы заносятся в двумерный массив. Предусмотреть возможность умножения матриц, а также вычисления матричной комплексной экспоненты $\exp(A) = E + A + \frac{A^2}{2!} + \dots + \frac{A^n}{n!} + \dots$, где A – комплексная матрица-аргумент, а E – единичная матрица.

Задача 16. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность складывать, вычитать и умножать комплексные числа. Создать производный класс для реализации квадратных матриц с комплексными элементами. Элементы матрицы заносятся в двумерный массив. Предусмотреть возможность умножения матриц, а также вычисления матричного комплексного косинуса $\cos(A) = E - \frac{A^2}{2!} + \frac{A^4}{4!} \dots + \frac{(-1)^n A^{2n}}{(2n)!} + \dots$, где A – комплексная матрица-аргумент, а E – единичная матрица.

Задача 17. Написать программу с классом для реализации комплексных чисел. Предусмотреть возможность складывать, вычитать и умножать комплексные числа. Создать производный класс для реализации квадратных матриц с комплексными элементами. Элементы матрицы заносятся в двумерный массив. Предусмотреть возможность умножения матриц, а также вычисления матричного комплексного синуса $\sin(A) = A - \frac{A^3}{3!} + \frac{A^5}{5!} \dots + \frac{(-1)^n A^{2n+1}}{(2n+1)!} + \dots$, где A – комплексная матрица-аргумент.

Задача 18. Написать программу с классом для реализации полиномиальных выражений $P_n(x) = \sum_{k=0}^n a_k x^k$ степени, не выше n (внешняя константа).

Определить операции сложения, вычитания полиномов, а также умножения и деления полиномов на число. Создать производный класс, в котором реализовать выражения вида $P_n(x) \exp(ax)$, где a – поле класса. Предусмотреть возможность вычисления производной от такого выражения – производная равна $(aP_n(x) + P'_n(x)) \exp(ax)$, где $P'_n(x) = \sum_{k=0}^{n-1} (k+1)a_{k+1}x^k$.

Задача 19. Написать программу с классом для реализации полиномиальных выражений $P_n(x) = \sum_{k=0}^n a_k x^k$ степени, не выше n (внешняя константа).

Определить операции сложения, вычитания полиномов, а также умножения и деления полиномов на число. Создать производный класс, в котором реализовать выражения вида $(ax + b)P_n(x)$, a и b – поля класса. Предусмотреть возможность вычисления производной от такого выражения – производная равна $(ax + b)P'_n(x) + aP_n(x)$, где $P'_n(x) = \sum_{k=0}^{n-1} (k+1)a_{k+1}x^k$.

Задача 20. Написать программу с классом для реализации комплексных чисел и основных операций с этими числами: сложения, вычитания, умножения и деления. Создать производный класс для работы с комплексными полиномиальными выражениями степени не выше, чем n . Выражение имеет вид $P_n(z) = \sum_{k=0}^n a_k z^k$, где z – комплексное число, а a_k – действительные числа (элементы массива производного класса). Предусмотреть возможность сложения и вычитания таких полиномов.

Глава 12

Шаблоны

Нормальные герои всегда идут в обход...

Из к/ф «Айболит 66»

Вне зависимости от используемого языка программирования, фундаментальным положением при составлении программного кода является алгоритм, который реализуется программой. Нередко при работе с данными разных типов прибегают к одним и тем же универсальным алгоритмам. Естественным образом возникает желание для подобных ситуаций разработать единый программный код с реализацией общего алгоритма, а затем использовать его для каждой конкретной ситуации. Обычно при этом возникают проблемы технического характера.

Например, представим, что необходимо разработать метод, у которого два аргумента, а действие метода состоит в том, что меняются местами значения аргументов. В принципе, здесь нет ничего сложного: аргументами являются переменные одного и того же типа (какого именно – вопрос отдельный). Аргументы передаются по ссылке (чтобы можно было изменить их значение). В теле метода вводится локальная переменная того же типа, что и тип аргументов. Переменной присваивается, например, значение первого аргумента, первому аргументу присваивается значение второго аргумента, а второму аргументу – значение локальной переменной. Все!

Очевидно, что указанный алгоритм не зависит от конкретного типа аргументов метода. Тем не менее, чтобы данная процедура могла выполняться с разными типами переменных, метод придется перегружать – для каждого типа данных пришлось бы создавать отдельный вариант метода. Это не тот путь, которым идут нормальные герои. Для таких случаев в C++ предусмотрено создание обобщенных функций и обобщенных классов.

Обобщенные функции

Обобщенная функция – это, по сути, функция, в которую тип данных, с которыми работает функция, передается в виде параметра. Определяется обобщенная функция практически так же, как и обычная функция, однако для обозначения типа локальных переменных и аргументов исполь-

зуется специальный идентификатор (или идентификаторы), введенный программистом. Впоследствии при вызове функции в соответствии с тем, какого типа аргументы передаются функции, автоматически определяется тип (или типы) данных, которые необходимо использовать в функции в том месте, где в коде функции использовался идентификатор типа.

Синтаксис объявления обобщенной функции с одним обобщенным типом (т.е. типом, который передается в функцию в виде параметра) имеет следующий вид:

```
template <class тип> тип_результата имя_функции(аргументы) {
    //код функции
}
```

Начинается описание обобщенной функции с ключевого слова `template`, после которого в угловых скобках размещается конструкция `<class тип>`, в которой после ключевого слова `class` указывается идентификатор `тип`, который является формальным обозначением типа данных. Далее, фактически, следует обычное описание функции, в котором, правда, в качестве типа данных можно использовать идентификатор `тип`.

Поясним сказанное на простом примере. В листинге 12.1 приведен программный код, в котором создается обобщенная функция с одним аргументом. Функция простая – все ее действие состоит в том, что на экран выводится значение аргумента.

Листинг 12.1. Обобщенная функция с одним аргументом

```
#include <iostream>
using namespace std;
//Обобщенная функция:
template <class X> void show(X arg){
    cout<<"Value is "<<arg<<endl;}
int main(){
    int n=5;
    double x=3.6;
    char s='a';
    //Вызов обобщенной функции:
    show(n);
    show(x);
    show(s);
    return 0;}
```

Прототип обобщенной функции имеет в данном случае вид `template <class X> void show(X arg)`. Это означает, что у функции один аргумент обобщенного типа `X`. Какой именно тип подразумевать под типом

данных, обозначенном как X, определяется в соответствии с тем, как вызывается обобщенная функция. Например, в главном методе программы объявлены три переменные: переменная n типа int, переменная x типа double и переменная s типа char.

При вызове функции командой `show(n)` в качестве типа данных X используется тип int, поскольку именно такого типа аргумент передан функции при вызове. Аналогично при вызове функции в формате `show(x)` в качестве типа X используется тип данных double, а для команды `show(s)` в качестве типа X используется тип char. В результате выполнения программы получим

```
Value is 5
Value is 3.6
Value is a
```

Приведенный пример исключительно прост. В нем по большому счету использовано только формальное объявление обобщенного типа. В листинге 12.2 показано, как в теле обобщенной функции создаются локальные переменные обобщенного типа.

Листинг 12.2. Создание локальной переменной обобщенного типа

```
#include <iostream>
using namespace std;
//Обобщенная функция:
template <class X> X AddOne(X arg){
    X t;
    t=arg+1;
    return t;}
int main(){
    int n=5;
    double x=3.6;
    char s='a';
    //Вызов обобщенной функции:
    cout<<n<<" + 1 = "<<AddOne(n)<<endl;
    cout<<x<<" + 1 = "<<AddOne(x)<<endl;
    cout<<s<<" + 1 = "<<AddOne(s)<<endl;
    return 0;}
```

В результате выполнения этой программы получим:

```
5 + 1 = 6
3.6 + 1 = 4.6
a + 1 = b
```

В программе определяется обобщенная функция `AddOne()`, у которой один аргумент обобщенного типа, а в качестве результата возвращается

значение аргумента, увеличенное на единицу. Поэтому тип возвращаемого функцией результата совпадает с типом аргумента. Если аргументом функции передается число типа `int` или `double`, результатом также является число типа `int` или `double` соответственно, на единицу большее аргумента. Если аргумент функции при вызове – символ (переменная типа `char`), в качестве значения возвращается следующий символ кодовой таблицы.

В обобщенных функциях может использоваться несколько обобщенных типов. Идентификаторы обобщенных типов в этом случае перечисляются через запятую в блоке в угловых скобках, и перед каждым из них указывается ключевое слово `class`. Пример использования обобщенной функции с двумя аргументами разных обобщенных типов приведен в листинге 12.3.

Листинг 12.3. Обобщенная функция с двумя аргументами

```
#include <iostream>
using namespace std;
//Обобщенная функция:
template <class X,class Y> void show(X x,Y y){
    cout<<"1-st argument: "<<x<<endl;
    cout<<"2-d argument: "<<y<<endl;}
int main(){
    //Вызов обобщенной функции:
    show(1, 'a');
    show("TEXT", 3.5);
    return 0;}
```

Функция достаточно проста: все ее назначение состоит в том, что на экран выводятся значения аргументов. В результате выполнения программы получаем:

```
1-st argument: 1
2-d argument: a
1-st argument: TEXT
2-d argument: 3.5
```

Обращаем внимание, что в данном случае аргументом функции можно передавать и текстовый литерал. Вообще же, какого типа аргументы допустимо передавать обобщенной функции при вызове, определяют исходя из программного кода обобщенной функции: использованные операции должны быть применимы к данным соответствующего типа. В данном случае функции `show()` допустимо передавать аргументы, значения которых могут выводиться на экран с помощью оператора вывода `<<`.

Перегрузка обобщенных функций

Вернейший путь к истине – познавать вещи, как они есть на самом деле, а не заключать о них так, как учили.

Д. Локк

Для обобщенных функций можно выполнять явную перегрузку, которая также называется **явной специализацией**. При явной перегрузке обобщенной функции создается отдельный вариант этой функции с явным указанием типов аргументов и результата функции. Явная специализация функции – это фактически стандартное описание обычной, не обобщенной, функции. Пример явной специализации обобщенной функции приведен в листинге 12.4.

Листинг 12.4. Явная специализация обобщенной функции

```
#include <iostream>
using namespace std;
//Обобщенная функция:
template <class X> void change(X &a,X &b){
    X t;
    t=a;
    a=b;
    b=t;}
//Явная специализация обобщенной функции:
void change(int &a,int &b){
    int t;
    t=a;
    a=b+1;
    b=t+1;}
int main(){
    double x=2.3,y=4.5;
    int m=6,n=8;
    //Вызов обобщенной функции:
    change(x,y);
    cout<<"x = "<<x<<endl;
    cout<<"y = "<<y<<endl;
    change(m,n);
    cout<<"m = "<<m<<endl;
    cout<<"n = "<<n<<endl;
    return 0;}
```

Обобщенная функция `change()` имеет два аргумента одного обобщенного типа, которые передаются по ссылке. В результате выполнения функции

происходит обмен значениями аргументов функции (именно для этого аргументы передаются по ссылке). Это происходит для аргументов всех типов, кроме аргументов типа `int`. На этот случай в программе создана явная специализация функции `change()`. Происходит не просто обмен значениями аргументов, а эти значения еще и увеличиваются на единицу. Поэтому в результате выполнения программы получаем следующий результат:

```
x = 4.5
y = 2.3
m = 9
n = 7
```

Таким образом, при вызове обобщенной функции реализуется один и тот же алгоритм во всех случаях, кроме ситуации, когда аргументы функции имеют тип `int`. В этом отношении явная специализация обобщенной функции позволяет создать своеобразное исключение из правила – правилом в данном случае служит основной вариант обобщенной функции.

Для обобщенных функций можно перегружать не только их конкретные реализации (т.е. создавать явные специализации), но перегружать целиком весь шаблон обобщенной функции. Перегрузка шаблона обобщенной функции осуществляется путем создания обобщенной функции с тем же названием, но измененным прототипом. Причем в данном случае изменение прототипа автоматически означает изменение количества аргументов обобщенной функции. Пример перегрузки шаблона обобщенной функции приведен в листинге 12.5.

Листинг 12.5. Перегрузка шаблона обобщенной функции

```
#include <iostream>
using namespace std;
//Обобщенная функция:
template <class X> X MySum(X a,X b){
    return a+b;}
//Перегрузка обобщенной функции:
template <class X> X MySum(X a){
    return a+1;}
int main(){
    int n=2,m=4;
    double x=4.2,y=3.4;
    char s='a';
    //Вызов обобщенной функции:
    cout<<MySum(n,m)<<endl;//Результат: 6
    cout<<MySum(x,y)<<endl;//Результат: 7.6
    cout<<MySum(s)<<endl;//Результат: b
    return 0;}
```

Программным кодом предусмотрено два варианта обобщенной функции `MySum()`: с одним и двумя аргументами обобщенного типа. Если у функции два аргумента, в качестве значения функции возвращается сумма аргументов. Если функции передается один аргумент, для вычисления значения функции к этому аргументу прибавляется единица.

Отметим также достаточно очевидное обстоятельство: при объявлении обобщенных функций кроме аргументов обобщенных типов могут использоваться и обычные аргументы. Для таких аргументов тип указывается в явном виде.

Обобщенные классы. Шаблоны

Концепция обобщенных функций естественным образом развивается и на случай классов. Несложно догадаться, что в C++ можно создавать классы, в которых тип данных передается как формальный параметр. Такие классы называются обобщенными, или просто шаблонами.

Объявление обобщенного класса начинается с ключевого слова `template`, после которого в угловых скобках с использованием ключевого слова `class` перечисляются формальные названия для типов данных. Во всем остальном объявление класса изменений не претерпевает, за исключением, пожалуй, того, что в качестве типов данных в полях и методах класса можно использовать формальные обозначения для типов данных, приведенные в шапке объявления. Пример объявления и использования обобщенного класса приведен в листинге 12.6.

Листинг 12.6. Обобщенный класс

```
#include <iostream>
using namespace std;
//Обобщенный класс:
template <class X> class MyClass{
    X value;
public:
    MyClass(X m){
        value=m;}
    void set(X m){value=m;}
    void get(){
        cout<<"value = "<<value<<endl;}
};
int main(){
    //Создание объекта с int-полем:
```

```

MyClass<int> a(5);
a.get();
a.set(3);
a.get();
//Создание объекта с char-полем:
MyClass<char> b('x');
b.get();
b.set('z');
b.get();
return 0;}

```

В результате выполнения этой программы получаем такие сообщения:

```

value = 5
value = 3
value = x
value = z

```

В программе объявлен обобщенный класс `MyClass` с прототипом `template <class X> class MyClass`. В соответствии с этим прототипом при описании полей и методов класса используется обобщенный тип `X`. Впоследствии при создании объекта класса `MyClass` необходимо будет указать, какой тип необходимо использовать в тех местах, где использован идентификатор `X`. Делается это путем указания соответствующего типа в угловых скобках после имени класса при объявлении соответствующего объекта. Например, инструкцией `MyClass<int> a(5)` создается объект `a` класса `MyClass`, в качестве обобщенного типа данных используется тип `int`, а конструктору класса при создании объекта передается целочисленный аргумент 5. Аналогично командой `MyClass<char> b('x')` создается объект `b` класса `MyClass`, вместо обобщенного типа использован тип данных `char`, аргумент конструктора – символ `'x'`.

У обобщенного класса `MyClass` закрытое поле `value` типа `X`, конструктор с одним аргументом того же типа и два метода: метод `set()` с аргументом типа `X` для изменения значения закрытого поля класса и метод `get()` без аргументов для отображения значения закрытого поля.

В главном методе программы на основе обобщенного класса создаются два объекта. Один объект имеет целочисленное поле, второй – символьное. Благодаря использованию единого универсального шаблона при создании объектов каждый объект имеет методы для работы с закрытым полем. Методы разных объектов имеют одинаковые названия и предназначены для выполнения однотипных функций. Такой подход существенно упрощает процесс создания программных кодов и делает их более компактными.

Типы по умолчанию и явная специализация класса

А у вас нет такого же, но без крыльев?

Из к/ф «Бриллиантовая рука»

С одной стороны, при создании объекта на основе обобщенного класса нет возможности автоматически определять значения для обобщенных типов, как это происходит в обобщенных функциях (на основе аргументов). Поэтому значения для обобщенных типов необходимо указывать в явном виде. С другой стороны, такой подход не очень удобен, поскольку приводит к усложнению процесса объявления объектов. В некоторых случаях разумно использовать значения для обобщенных типов, используемые по умолчанию.

Речь идет о следующей ситуации. Создается обобщенный класс, в котором для обобщенных типов предусмотрены значения по умолчанию. Если затем при создании объекта обобщенный тип данных явно не указан (хотя угловые скобки все равно придется указать), то используется значение типа по умолчанию. Для того чтобы задать обобщенный тип по умолчанию, необходимо в прототипе класса в том месте, где вводится индикатор обобщенного типа, указать через знак равенства тип по умолчанию. Пример приведен в листинге 12.7.

Листинг 12.7. Значение обобщенного типа данных по умолчанию

```
#include <iostream>
using namespace std;
//Обобщенный класс:
template <class X=int> class MyClass{
    X value;
public:
    MyClass(X m){
        value=m; }
    void set(X m){value=m; }
    void get(){
        cout<<"value = "<<value<<endl; }
};
int main(){
    //Создание объекта с int-полем:
    MyClass<> a(5);
    a.get();
    a.set(3);
    a.get();
```

```
//Создание объекта с char-полем:
MyClass<char> b('x');
b.get();
b.set('z');
b.get();
return 0;}
```

Здесь представлен несколько видоизмененный программный код, рассмотренный ранее (см. листинг 12.6). Фактически изменен прототип объявления обобщенного класса: вместо инструкции `<class X>` использована инструкция `<class X=int>`, которая означает, что если при создании объекта тип `X` явно не указан, следует в качестве `X` использовать тип `int`. Поэтому создать объект с использованием типа `int` теперь можно, например, командой `MyClass<> a(5)`. В остальном как код, так и результат выполнения программы неизменны.

Как и для обобщенных функций, для обобщенных классов можно создавать явные специализации. Явная специализация класса позволяет создать исключение из общего правила, определяемого шаблоном обобщенного класса. При создании явной специализации обобщенного класса после ключевого слова `template`, с которого начинается описание специализации, следуют пустые угловые скобки. Далее следует ключевое слово `class`, название класса и, в угловых скобках, тип (или типы) данных, для которых выполняется явная специализация. Пример создания явной специализации класса приведен в листинге 12.8.

Листинг 12.8. Явная специализация обобщенного класса

```
#include <iostream>
using namespace std;
//Обобщенный класс:
template <class X> class MyClass{
    X value;
public:
    MyClass(X m){
        value=m;}
void set(X m){value=m;}
void get(){
    cout<<"value = "<<value<<endl;}
};
//Явная специализация обобщенного класса:
template <> class MyClass<int>{
public:
    int value;
    MyClass(){
        value=5;}
};
```

```
int main() {
    //Создание объекта с int-полем:
    MyClass<int> a;
    cout<<"value = "<<a.value<<endl;
    a.value=3;
    cout<<"value = "<<a.value<<endl;
    //Создание объекта с char-полем:
    MyClass<char> b('x');
    b.get();
    b.set('z');
    b.get();
    return 0;}

```

При описании явной специализации класса использован прототип `template<> class MyClass<int>`. Это означает, что явная специализация создается для случая, когда в качестве обобщенного типа данных используется `int`-тип. В приведенной явной специализации поле `value` объявлено как открытое, конструктор класса в этом случае не имеет аргумента (при создании объекта полю `value` присваивается значение 5), а также упразднены методы `set()` и `get()`. Поэтому для соответствующих объектов обращение к полю `value` выполняется в явном виде (что и сделано в главном методе программы). Результат выполнения программы такой же, как и программы из листинга 12.6. Что касается создания объектов на основе шаблонных классов, то оно выполняется так же, как и в предыдущих случаях: после имени класса необходимо в угловых скобках указать значение типа данных. Если для указанного типа явная специализация не предусмотрена, используется общий шаблон, если специализация предусмотрена – шаблон для специализации.

При создании обобщенных классов может использоваться сразу несколько обобщенных типов данных и(или) стандартные типы (т.е. типы данных, указанные в явном виде). Пример такой ситуации приведен в листинге 12.9.

Листинг 12.9. Несколько типов в обобщенном классе

```
#include <iostream>
#include <cstdlib>
using namespace std;
//Обобщенный класс:
template <class X,class Y,int n> class MyClass{
public:
    //Поля класса - массивы:
    X xarray[n];
    Y yarray[n];
    //Конструктор класса:
    MyClass() {

```

```

//Инициализация генератора случайных чисел:
srand(1);
//Заполнение массивов случайными значениями:
for(int i=0;i<n;i++){
    xarray[i]=(X) (100+rand()%10);
    yarray[i]=(Y) (100+rand()%10);
}
//Методы для отображения массивов:
void getx(){
    for(int i=0;i<n;i++) cout<<xarray[i]<<" ";
    cout<<endl;}
void gety(){
    for(int i=0;i<n;i++) cout<<yarray[i]<<" ";
    cout<<endl;}
};
int main(){
    //Размеры массивов-полей класса:
    const int n=3,m=5;
    //Создание объектов:
    MyClass<int,char,n> a;
    MyClass<char,double,m> b;
    //Проверка значений полей объектов:
    a.getx();
    a.gety();
    b.getx();
    b.gety();
    return 0;}

```

В результате выполнения программы получаем:

```

101 104 109
k d h
e h m l f
107 100 104 108 104

```

В программе создается обобщенный класс `MyClass`, прототип которого имеет вид `template <class X,class Y,int n> class MyClass`. Описание класса содержит два обобщенных типа `X` и `Y`. Также в прототипе класса указана переменная `n` стандартного типа `int`. Класс содержит два поля, которые являются массивами типов `X` и `Y` соответственно, а размер каждого из этих массивов задается значением переменной `n`. В классе описан конструктор, которым при создании объекта элементы полей-массивов заполняются случайными значениями. В конструкторе использована функция `srand()` для инициализации генератора случайных чисел (чтобы каждый раз при запуске программы генерировалась одна и та же псевдослучайная последовательность чисел). Значения элементов массивов определя-

ются в рамках оператора цикла командами (X) $(100 + \text{rand}() \% 10)$ и (Y) $(100 + \text{rand}() \% 10)$. В данном случае использованы инструкции (X) и (Y) приведения целочисленного результата к типу X и Y соответственно, поскольку командой $100 + \text{rand}() \% 10$ возвращается случайное целое число в диапазоне от 100 до 109 включительно, а элементы массивов имеют тип X и Y. В классе MyClass предусмотрены методы `getx()` и `gety()` для отображения (в одну строку) значений элементов массивов-полей класса.

В главном методе программы командами `MyClass<int, char, n> a` и `MyClass<char, double, m> b` создаются объекты a и b. При создании объектов указываются значения для обобщенных типов данных, а также целочисленные значения, определяющие размеры массивов – в данном случае это константы m и n. В результате объект a имеет два поля: массив `xarray` из n=3 элементов типа `int` и массив `yarray` из n=3 элементов типа `char`. У объекта b есть поле-массив `xarray` из m=5 элементов типа `char` и поле-массив `yarray` из m=5 элементов типа `double`. Обращение к методам объектов для отображения значения полей выполняется стандартным способом.

Примеры решения задач

Здесь рассматриваются некоторые примеры создания программных кодов, в которых используются шаблоны.

■ Обобщенная экспонента

Создадим обобщенную функцию, с помощью которой можно было бы вычислять значение экспоненты как от действительного числа, так и для комплексного аргумента. Комплексные числа будем реализовывать с помощью класса пользователя. Особенность процедуры вычисления состоит в том, что экспонента рассчитывается в виде ряда Тейлора. Весь программный код приведен в листинге 12.10.

Листинг 12.10. Обобщенная экспонента

```
#include <iostream>
const N=100;
using namespace std;
//Класс для реализации комплексных чисел:
class Complex{
public:
    double Re, Im;
    //Перегрузка оператора присваивания:
    Complex operator=(double x){
```



```

    Re=x;
    Im=0;
    return *this;}
//Перегрузка оператора умножения:
Complex operator*(Complex obj){
    Complex tmp;
    tmp.Re=Re*obj.Re-Im*obj.Im;
    tmp.Im=Re*obj.Im+Im*obj.Re;
    return tmp;}
//Перегрузка оператора сложения:
Complex operator+(Complex obj){
    Complex tmp;
    tmp.Re=Re+obj.Re;
    tmp.Im=Im+obj.Im;
    return tmp;}
//Перегрузка оператора деления:
Complex operator/(double x){
    Complex tmp;
    tmp.Re=Re/x;
    tmp.Im=Im/x;
    return tmp;}
};
//Шаблонная функция для определения экспоненты:
template <class X> X mExp(X t){
    X s;
    s=1;
    X q=t;
    int i;
    for(i=1;i<=N;i++){
        s=s+q;
        q=q*t/(i+1);
    }
    return s;}
int main(){
    Complex z;
    z.Re=1;
    z.Im=2;
    cout<<"exp("<<z.Re<<"+"<<z.Im<<"i) ="<<mExp(z).Re<<"+"
    <<mExp(z).Im<<"i"<<endl;
    cout<<"exp(1.0) ="<<mExp(1.0)<<endl;
    cout<<"exp(1) ="<<mExp(1)<<endl;
    return 0;}

```

При вычислениях использовано выражение для экспоненты $\exp(z) \approx 1 + z + \frac{z^2}{2!} + \dots + \frac{z^N}{N!}$, где в общем случае аргумент z является

комплексным числом. Итерационная процедура по вычислению ряда реализуется с помощью шаблонной функции `mExp()`. Аргументом функции может быть числовое значение или объект класса `Complex`, описанного в начале программы. Помимо двух полей, класс содержит несколько перегруженных операторных функций для того, чтобы можно было множить и складывать комплексные числа, делить их на число и присваивать в качестве значения действительные числа.

Результат выполнения программы имеет следующий вид:

```
exp(1+2i)=-1.1312+2.47173i
exp(1.0)=2/71828
exp(1)=2
```

Обращаем внимание читателя, что в случае, если аргументом функции указано целое число, то результатом также является число целое. Проблема связана с тем, что для целочисленных операндов по умолчанию операция деления выполняется как деление нацело. Предлагаем читателю самостоятельно предложить возможные пути ее решения.

■ Перегрузка операторов

Рассмотрим пример с перегрузкой операторных функций в обобщенных классах. Пример достаточно простой: в программе объявляются два разных класса `A` и `B`. Класс `A` имеет целочисленное поле `k`, а класс `B` имеет поля `x` и `y` типа `double`. В каждом из классов перегружается оператор сложения. Оператор сложения перегружается так, что в результате сложения двух объектов получаем объект того же класса. При этом соответствующие поля объектов складываются. Каждый класс имеет метод `show()` для отображения значения поля или полей. Также в программе объявляется обобщенный класс `MyTempl`, в котором имеется одно поле `res` (объект класса `A` или `B`). В обобщенном классе `MyTempl` также перегружается оператор сложения: при сложении объектов класса `MyTempl` складываются их поля `res`. Для отображения значений полей объекта-поля `res` используется метод `show()`. При его реализации вызывается метод с таким же именем из объекта `res`. Программный код приведен в листинге 12.11.

Листинг 12.11. Перегрузка операторов

```
#include <iostream>
using namespace std;
class A{
public:
    int k;
    A operator+(A obj){
```

```

    A tmp;
    tmp.k=k+obj.k;
    return tmp;}
void show(){
    cout<<"k = "<<k<<endl;}
};
class B{
public:
double x,y;
B operator+(B obj){
    B tmp;
    tmp.x=x+obj.x;
    tmp.y=y+obj.y;
    return tmp;}
void show(){
    cout<<"x = "<<x<<endl;
    cout<<"y = "<<y<<endl;}
};
template <class X> class MyTempl{
public:
X res;
MyTempl operator+(MyTempl obj){
    MyTempl tmp;
    tmp.res=res+obj.res;
    return tmp;}
void show(){
    res.show();}
};
int main(){
    MyTempl<A> a1,a2,a3;
    MyTempl<B> b1,b2,b3;
    a1.res.k=1;
    a2.res.k=2;
    b1.res.x=10.1;
    b1.res.y=100.1;
    b2.res.x=20.2;
    b2.res.y=200.2;
    a3=a1+a2;
    b3=b1+b2;
    a3.show();
    b3.show();
    return 0;}

```

В результате выполнения программы получаем:

```

k = 3
x = 30.3
y = 300.3

```

Стоит отметить, что поскольку в обобщенном классе метод `show()` реализован через вызов метода `show()` из объекта `res`, в качестве типа-параметра `X` можно использовать только имена классов, в которых объявлен такой метод (и, разумеется, перегружен оператор сложения).

■ Перестановка элементов массива

В следующем примере создается обобщенный класс с полем-массивом, тип элементов которого является параметром обобщенного класса. В обобщенном классе перегружен оператор `[]` так, что с его помощью можно обращаться к элементам поля-массива (причем при выходе за пределы массива выполняется циклический переход в начало массива), а также оператор инкремента, действие которого сводится к циклической перестановке элементов в массиве (первый элемент смещается на место второго, второй смещается на место третьего и т.д., а последний элемент переходит на место первого элемента). В классе также описывается метод `show()` для отображения элементов поля-массива. Программный код приведен в листинге 12.12.

Листинг 12.12. Перестановка элементов массива

```
#include <iostream>
using namespace std;
//Размер поля-массива:
const int n=10;
template <class X> class MyClass{
public:
    //Поле-массив:
    X array[n];
    //Перегрузка оператора []:
    X operator[](int k){
        return array[k%n];}
    //Перегрузка оператора ++:
    MyClass operator++(){
        X tmp;
        int i;
        tmp=array[n-1];
        for(i=n-1;i>0;i--)
            array[i]=array[i-1];
        array[0]=tmp;
        return *this;}
    //Метод для отображения массива:
    void show(){
        for(int i=0;i<n;i++)
            cout<<array[i]<<" ";
```

```

        cout<<endl;}
//Конструктор класса:
MyClass() {
    for(int i=0;i<n;i++)
        array[i]=(X) (rand() %25+100);
    }
};

int main(){
    MyClass<int> obj1;
    obj1.show();
    for(int i=n;i<2*n;i++)
        cout<<obj1[i]<<" ";
    cout<<endl;
    MyClass<char> obj2;
    obj2.show();
    ++obj2;
    obj2.show();
    return 0;}

```

Результат выполнения программы может иметь следующий вид:

```

116 117 109 100 119 124 103 108 112 114
116 117 109 100 119 124 103 108 112 114
i x j f o t x u f o
o i x j f o t x u f

```

При создании объекта вызывается конструктор для случайного заполнения массива-поля создаваемого объекта. Массив заполняется следующим образом: генерируется случайное целое число, после чего это значение преобразуется к типу, который имеют элементы массива.

В главном методе программы проиллюстрирован процесс создания объектов с разным типом-параметром, индексация объектов, результат указания индекса за пределами размера массива, а также циклическая перестановка элементов массива.

■ Поиск совпадений

Рассмотрим задачу о поиске совпадений некоторого значения со значениями элементов массива. Массив реализуется в виде поля обобщенного класса. Для поиска совпадений создается обобщенная функция, первым аргументом которой указывается проверяемое на предмет совпадения значение, а второй аргумент – объект с массивом, в котором выполняется поиск. В результате вызова функции на экран выводится сообщение о количестве совпадений. Соответствующий программный код приведен в листинге 12.13.

Листинг 12.13. Поиск совпадений

```

#include <iostream>
using namespace std;
const int n=20;
template <class X> class Elements{
public:
    X array[n];
    X operator[](int k){
        return array[k];}
    Elements(){
        for(int i=0;i<n;i++){
            array[i]=(X) (rand()%20+100);
            cout<<array[i]<<" ";}
        cout<<endl;}
};
template <class X> void FindElement(X s,Elements<X> obj){
    int count=0,i;
    for(i=0;i<n;i++)
        if(s==obj[i]) count++;
    cout<<"result is "<<count<<endl;}
int main(){
    Elements<int> a;
    FindElement(101,a);
    Elements<char> b;
    FindElement('f',b);
    return 0;}

```

В обобщенном классе `Elements` перегружается оператор `[]` для индексации объектов. При создании объекта элементы массива заполняются с помощью генератора случайных чисел с одновременным выводом значений на экран.

Как отмечалось, у обобщенной функции `FindElement()` два аргумента, причем по типу первого аргумента определяется тип параметра для объекта класса `Elements`, передаваемого вторым аргументом функции. Поиск совпадений осуществляется путем последовательного перебора элементов массива, являющегося полем объекта – второго аргумента функции `FindElement()`.

В главном методе программы на основе обобщенного класса `Elements` создаются два объекта. На их основе иллюстрируются методы вызова функции `FindElement()`. В результате выполнения программы получаем, например, следующее:

```

101 107 114 100 109 104 118 118 102 104 105 105 101 107 101
111 115 102 107 116
result is 3

```

```
o h f q p f e t v s k j o v m p k w r s
result is 2
```

Отметим, что перегрузка оператора [] в данном случае выполнялась исключительно ради удобства.

■ Наследование шаблона

В листинге 12.14 приведен пример программного кода, в котором использовано наследование обобщенного класса.

Листинг 12.14. Наследование шаблона

```
#include <iostream>
using namespace std;
//Базовый обобщенный класс:
template <class X> class First{
public:
    X a;
    First(X arg){
        a=arg;}
};
//Производный обобщенный класс:
template <class X,class Y> class Second:public First<X>{
public:
    Y b;
    void show(){
        cout<<"a = "<<a<<endl;
        cout<<"b = "<<b<<endl;}
    Second(X arg1,Y arg2): First<X>(arg1){
        b=arg2;}
};
int main(){
    Second<int,char> obj(5,'z');
    obj.show();
    return 0;}
```

Пример очень простой. Сначала создается обобщенный класс `First`, который имеет один тип-параметр, поле и конструктор с одним аргументом. Аргумент конструктора при создании объекта присваивается в качестве значения полю.

Класс-наследник `Second` имеет два параметра типа и создается на основе базового обобщенного класса `First`. Обращаем внимание, что при описании механизма наследования базовый класс указывается с формальным параметром типа.

В производном классе добавляется еще одно поле, а также описывается конструктор с двумя аргументами. Как и при описании механизма наследования, при передаче аргумента конструктору базового класса указывается также и параметр типа.

Результат выполнения программы имеет вид

```
a = 5
b = z
```

Результат отображается с помощью метода `show()`, описанного в производном классе.

■ Создание обобщенного дерева

Рассмотрим программу, в которой создается обобщенный класс. С помощью объектов этого класса в главном методе программы формируется структура древовидного типа. Аналогичные задачи уже рассматривались ранее, и в частности, в главе 9. Здесь использовавшийся в главе 9 подход несколько модифицирован и расширен, для того, чтобы можно было создавать деревья, во-первых, с разным количеством веток, а во-вторых, с полями разных типов. Рассмотрим программный код, приведенный в листинге 12.15.

Листинг 12.15. Обобщенное дерево

```
#include <iostream>
using namespace std;
//Количество веток дерева:
const int n=2;
//Обобщенный класс:
template <class X> class MyTree{
public:
//Обобщенное поле:
X record;
//Массив указателей:
MyTree *pnt[n];
//Конструктор:
MyTree(int k){
    int i;
    if(k==1)
        for(i=0;i<n;i++) pnt[i]=NULL;
    else
        for(i=0;i<n;i++) pnt[i]=new MyTree(k-1);
    cout<<"created: "<<this<<endl;
```



```

    for(i=0;i<n;i++)
        cout<<i<<": "<<pnt[i]<<endl;
    cout<<endl;}
//Деструктор:
~MyTree() {
    int i;
    for(i=0;i<n;i++)
        delete pnt[i];
    cout<<"deleted: "<<this<<endl;
}
};
int main() {
    //Указатель на первый объект дерева:
    MyTree<double> *input;
    //Создание дерева:
    input=new MyTree<double>(3);
    //Обращение к полю record объекта дерева:
    input->pnt[1]->record=3.1415;
    cout<<endl;
    cout<<input->pnt[1]<<": record=";
    cout<<(*(*input).pnt[1]).record<<endl;
    cout<<endl;
    //Удаление дерева:
    delete input;
    return 0;
}

```

Обобщенный класс `MyTree` имеет один обобщенный параметр – тип открытого поля `record`. Кроме этого поля, класс содержит поле-массив `pnt` указателей на объекты класса `MyTree`. Через эти указатели устанавливается связь между объектами дерева. Размер массива определяется через глобальную константу `n`. Эта константа фактически определяет количество веток дерева.

Вся работа по созданию дерева реализуется через конструктор класса `MyTree`. Конструктору передается целочисленный аргумент, который определяет количество уровней создаваемого дерева. При создании объектов выводится сообщение с адресом созданного объекта и значениями полей-указателей этого объекта (адреса тех объектов, на которые ссылается через свои указатели данный объект). Реализован конструктор через рекурсию: при создании объекта с аргументом конструктора `k` вызывается конструктор с аргументом конструктора `k-1`. При единичном аргументе создается всего один объект.

При вызове деструктора сначала удаляются все объекты, на которые ссылается удаляемый объект. Таким образом реализуется механизм удаления

всей ветки дерева, если удаляется узловой объект – в противном случае ссылки на эти объекты были бы потеряны и впоследствии доступ к ним отсутствовал бы. При удалении объекта также выводится сообщение с адресом удаляемого объекта.

В главном методе программы создается бинарное дерево из трех уровней (начальный элемент является первым уровнем дерева). Результат выполнения программы может иметь следующий вид:

```
created: 00355DC8
0: 00000000
1: 00000000

created: 00355EB0
0: 00000000
1: 00000000

created: 00355D78
0: 00355DC8
1: 00355EB0

created: 00356128
0: 00000000
1: 00000000

created: 00356178
0: 00000000
1: 00000000

created: 00355F00
0: 00356128
1: 00356178

created: 00355D28
0: 00355D78
1: 00355F00

00355F00: record=3.1415

deleted: 00355DC8
deleted: 00355EB0
deleted: 00355D78
deleted: 00356128
deleted: 00356178
deleted: 00355F00
deleted: 00355D28
```

Стоит обратить внимание на способ обращения к полям объектов в дереве. Например, инструкцией `input->pnt[1]->record` выполняется обращение к полю `record` объекта, размещенного по адресу `pnt[1]`, – адрес этого объекта записан в поле `pnt[1]` объекта с адресом `input`. Доступ к тому же самому полю может быть получен и с помощью команды `(*input).pnt[1].record`: ведь `(*input).pnt[1]` – это указатель на объект, к полю `record` которого выполняется обращение, сам объект представляется как `(*input).pnt[1]`. Следовательно, обращение к полю этого объекта может быть записано как `(*input).pnt[1].record`.

В заключение отметим то преимущество, которое получаем в результате использования обобщенного класса при создании дерева. Связано оно в первую очередь с тем, что формирование дерева на основе объектов обобщенного класса позволяет создавать, используя один и тот же код для класса, структуры с разным типом поля `record` – именно это поле на практике используется для записи «полезной» информации. Благодаря этому реализуется принцип универсальности базового алгоритма: вне зависимости от типа выстраиваемых в виде дерева базовых элементов (числа, символы или объекты) алгоритм создания дерева один и тот же.

Резюме

1. Обобщенная функция – это функция, в которую тип данных, с которыми работает функция, передается в виде параметра.
2. Начинается описание обобщенной функции с ключевого слова `template`. В угловых скобках после ключевого слова `class` указывается формальный идентификатор типа данных. Далее следует обычное описание функции, в котором в качестве типа данных можно использовать указанный идентификатор типа.
3. Для обобщенных функций можно выполнять явную перегрузку, которая также называется явной специализацией. При явной перегрузке обобщенной функции создается отдельный вариант этой функции с явным указанием типов аргументов и результата функции.
4. Для обобщенных функций можно перегружать не только их конкретные реализации (т.е. создавать явные специализации), но перегружать целиком весь шаблон обобщенной функции. Перегрузка шаблона обобщенной функции осуществляется путем создания обобщенной функции с тем же названием, но измененным прототипом.

5. Обобщенными классами, или шаблонами, называются классы, в которых тип данных передается как формальный параметр.
6. Объявление обобщенного класса начинается с ключевого слова `template`, после которого в угловых скобках с использованием ключевого слова `class` перечисляются формальные названия для типов данных. Во всем остальном объявление класса изменений не претерпевает. В качестве типов данных в полях и методах обобщенного класса можно использовать формальные обозначения для типов данных, приведенные в шапке объявления.
7. При создании обобщенных классов допускается для обобщенных типов определять значения по умолчанию.
8. Как и для обобщенных функций, для обобщенных классов можно создавать явные специализации. При создании явной специализации обобщенного класса после ключевого слова `template` следуют пустые угловые скобки, далее ключевое слово `class`, название класса и, в угловых скобках, тип (или типы) данных, для которых выполняется явная специализация.

Контрольные вопросы

Маленькая ложь рождает большое недоверие.

Из к/ф «Семнадцать мгновений весны»

1. Что такое обобщенная функция?
2. Как объявляется обобщенная функция?
3. Что такое явная перегрузка обобщенной функции и как она выполняется?
4. Что такое перегрузка шаблона обобщенной функции и как она выполняется?
5. Что такое обобщенный класс?
6. Как объявляется обобщенный класс?
7. Что такое значение по умолчанию для обобщенного типа?
8. Что такое явная специализация обобщенного класса и как она выполняется?

Задачи для самостоятельного решения

Далее приведен список задач, которые предлагается решить читателю самостоятельно. При решении рекомендуется использовать обобщенные функции и классы.

Задача 1. Написать программу с обобщенной функцией и аргументом-массивом. В результате выполнения функции на экран выводятся элементы массива.

Задача 2. Написать программу с обобщенной функцией и аргументом-массивом. В результате выполнения функции меняются местами элементы, индексы которых переданы аргументами функции.

Задача 3. Написать программу с обобщенной функцией и аргументом-массивом. В результате выполнения функции выполняется циклическая перестановка элементов массива.

Задача 4. Написать программу с обобщенной функцией и аргументом-массивом. В результате выполнения функции элементы массива размещаются в обратном порядке.

Задача 5. Написать программу с обобщенной функцией для подсчета выпадающих элементов массива – аргумента функции.

Задача 6. Создать класс для работы с комплексными числами и обобщенную функцию для вычисления косинуса от комплексного и действительного чисел. Косинус вычислять по формуле

$$\cos(z) = 1 - \frac{z^2}{2!} + \frac{z^4}{4!} - \dots + \frac{(-1)^n z^{2n}}{(2n)!} + \dots$$

Задача 7. Создать класс для работы с комплексными числами и обобщенную функцию для вычисления синуса от комплексного и действительного чисел. Синус вычислять по формуле

$$\sin(z) = z - \frac{z^3}{3!} + \frac{z^5}{5!} - \dots + \frac{(-1)^n z^{2n+1}}{(2n+1)!} + \dots$$

Задача 8. Создать класс для работы с комплексными числами и обобщенную функцию $\exp(-z) = 1 - z + \frac{z^2}{2!} - \frac{z^3}{3!} + \dots + \frac{(-1)^n z^n}{n!} + \dots$ для вычисления значения от действительного и комплексного аргумента.

Задача 9. Написать программу с обобщенным классом, у которого есть поле-массив. Перегрузить оператор сложения так, чтобы при добавлении к объекту класса целого числа выполнялся циклический сдвиг элементов массива на соответствующее количество позиций (вправо для положительного целочисленного операнда и влево для отрицательного).

Задача 10. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Перегрузить оператор `[]` так, чтобы можно было индексировать объекты обобщенного класса.

Задача 11. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Перегрузить оператор инкремента так, чтобы все строки в массиве циклически смещались на одну позицию.

Задача 12. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Перегрузить оператор инкремента так, чтобы все столбцы в массиве циклически смещались на одну позицию.

Задача 13. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Перегрузить оператор инкремента так, чтобы в префиксной форме действие оператора сводилось к тому, чтобы все строки в массиве циклически смещались на одну позицию, а в постфиксной форме на одну позицию смещались столбцы.

Задача 14. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Описать метод, с помощью которого меняются местами две строки массива. Индексы строк массива передаются аргументами метода.

Задача 15. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Описать метод, с помощью которого меняются местами два столбца массива. Индексы столбцов массива передаются аргументами метода.

Задача 16. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Описать метод, с помощью которого выполняется поиск элемента в массиве. В качестве результата возвращается общее количество совпадений (количество вхождений элемента в массив).

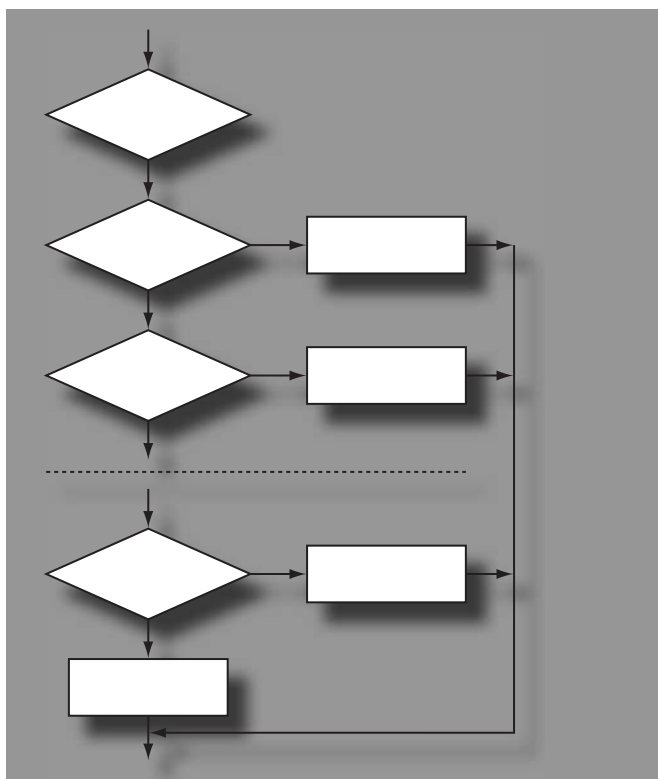
Задача 17. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Описать метод, с помощью которого выполняется построчный поиск элемента в массиве. В результате выводятся индексы элементов, совпадающие с элементом поиска.

Задача 18. Написать программу с обобщенным классом, у которого есть поле — двумерный массив. Описать метод, с помощью которого выполняется построчный вывод на экран подмассива, определяемого по левому верхнему и правому нижнему элементам. Индексы этих элементов передаются аргументами методу.

Задача 19. Написать программу для создания, на основе конструктора обобщенного класса, бинарного дерева объектов. В такой структуре каждый объект ссылается на два объекта такого же типа. Каждый из этих объектов, в свою очередь, ссылается на два объекта и т.д.

Задача 20. Написать программу для создания, на основе конструктора обобщенного класса, триарного дерева объектов. В такой структуре каждый объект ссылается на 3 объекта такого же типа. Каждый из этих объектов, в свою очередь, ссылается на 3 объекта и т.д.

Среды разработки



ЧАСТЬ III

Глава 13. Компиляторы и интегрированные среды разработки

Они - хорошие люди. Просто не продумали ситуацию до конца...

из к/ф "Плутовство"

Мало знать, как правильно составить программный код. Необходимо также четко представлять, где и как его набрать и, самое главное, что потом с ним делать. Обычно программный код набирается, компилируется, и если компиляция прошла успешно, программа запускается на выполнение. Хотя жесткого ограничения нет, но набирать код лучше все же в специальном редакторе – как правило, имеют режимы специального выделения блоков и ключевых слов программного кода, а нередко снабжены еще и встроенной автоматической проверкой синтаксиса. Поэтому разумно использовать редактор кодов. Кроме того, как правило, компилятор, редактор и прочие полезные утилиты поставляются в одном комплекте. Такой пакет услуг из серии "все включено" называется интегрированной средой разработки (сокращенно **IDE** от английского **Integrated Development Environment**). Условно все интегрированные среды разработки можно разделить на плохие и хорошие, дорогие и **бесплатные**. Причем **бесплатный** не всегда означает *плохой*, а *дорогой* не всегда означает *хороший*. К сожалению, основные хорошие бесплатные интегрированные среды разработки нередко поставляются без встроенных компиляторов. То есть в этом случае получается, что "не все включено", и кое-что, кроме среды разработки, приходится еще устанавливать. В этом приложении дается краткий обзор основных на сегодняшний день интегрированных сред разработки. В следующем приложении отдельно описывается распространяемый свободно пакет (среда разработки) Dev-C++.

Компиляторов вообще и коммерческих компиляторов на сегодня достаточно много. При их выборе важное значение имеет не только и не столько удобство работы с ними (и стоимость, если речь идет о коммерческих проектах), но и то, насколько данный компилятор поддерживает стандарты языка C++. Это важное условие, которое напрямую сказывается на совместимости программных кодов, написанных с помощью разных средств разработки и откомпилированных различными компиляторами. Среди наиболее надежных, эффективных и пользующихся спросом компиляторов и ин-

тегрированных сред разработки есть как коммерческие, так и бесплатные, свободно распространяемые.

Что касается коммерческих программных продуктов, то достаточно популярным на сегодня продуктом является интегрированная среда разработки Borland C++ Builder. Однако по популярности она, пожалуй, все же уступает приложению Visual C++ - продукту компании Microsoft, который является составной частью пакета Visual Studio. Следует отметить, что, кроме коммерческого продукта Visual C++, компанией Microsoft предлагаются для бесплатного использования продукты серии Express Edition, среди которых и приложение Visual C++ Express Edition, с помощью которого можно создавать, компилировать и выполнять программные коды.

Наиболее популярными и эффективными интегрированными средами разработки среди свободно распространяемых являются, пожалуй, Eclipse, NetBeans и Dev-C++. Все они по большому счету используют общие наборы компиляторов и отладчиков. Среда Dev-C++ свободно загружается через Internet. Желающие могут обратиться за дополнительной информацией к сайту www.bloodshed.net. Устанавливается Dev-C++ сразу с компилятором, так что особых проблем ни при установке, ни при работе не возникает. Необходимую для работы с Eclipse информацию, равно как и бесплатно загружаемые утилиты, можно найти на сайте www.eclipse.org. Наконец, поддержка проекта NetBeans осуществляется через сайт www.netbeans.org.

Что касается Eclipse и NetBeans, то по сравнению с Dev-C++ ситуация несколько сложнее. Во-первых, компиляторы для Eclipse и NetBeans необходимо устанавливать отдельно, и процедура эта далеко не самая тривиальная. Кроме того, для использования Eclipse и NetBeans предварительно необходимо установить виртуальную машину Java. Хотя все необходимые утилиты для Java имеются в свободном доступе (например, на сайте www.java.com), а в свободном доступе существуют неплохие компиляторы для C++ (например, GCC – сокращение от GNU C Compiler), процедуру установки и настройки оболочек Eclipse и NetBeans не назовешь слишком простой. Далее кратко описываются особенности работы с некоторыми из перечисленных программных продуктов. Внимание уделяется трем ключевым моментам:

1. Особенности создания с помощью данного компилятора/среды консольного приложения.
2. Наличие специфических для данного компилятора/среды заголовков или команд, которые используются при составлении программного кода.
3. Процессу компиляции и запуска программы на выполнение.

Эти вопросы рассматриваются очень кратко, но так, чтобы читатель смог составить общее представление о методах работы с компилятором/средой.

В первую очередь стоит обратить внимание на программные продукты, предлагаемые компанией Microsoft (сайт компании www.microsoft.com). На рис. 13.1 представлено рабочее окно приложения Microsoft Visual C++.

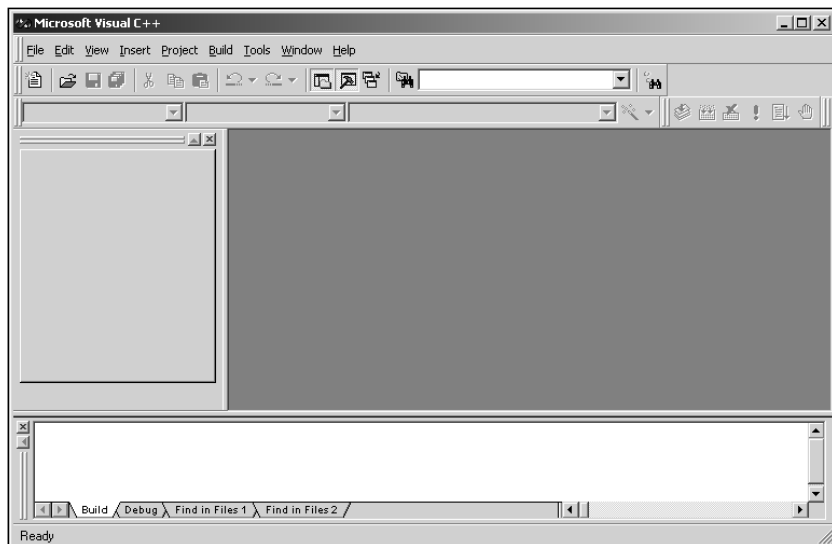


Рис. 13.1. Окно приложения Microsoft Visual C++

Для создания консольного приложения с помощью приложения Microsoft Visual C++ выбирают команду **File ► New**, в результате чего открывается окно **New**, представленное на рис. 13.2.

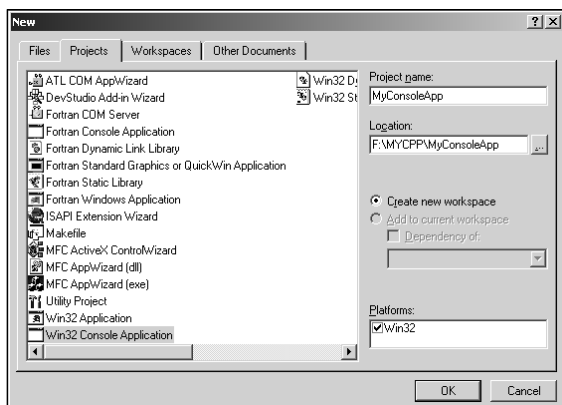


Рис. 13.2. Диалоговое окно **New** открыто на вкладке **Projects**

Окно следует открыть на вкладке **Projects**, в белом центральном списке выбрать пункт **Win32 Console Application**, в поле **Location** выбирается папка для сохранения проекта, а в поле **Project name** вводится имя создаваемого проекта. После этого щелкают кнопку **OK**. В результате открывается еще одно окно мастера создания консольного приложения, представленное на рис. 13.3.

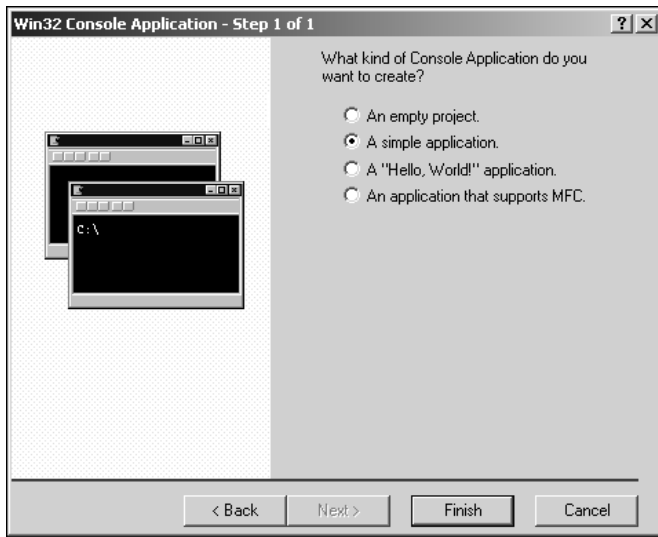


Рис. 13.3. Окно мастера создания консольного приложения

Окно содержит группу переключателей из четырех положений. Разумно выбрать пункт **A simple application** (если, разумеется, не нужно создать, например, приложение с поддержкой библиотеки классов MFC – сокращение от Microsoft Foundation Classes). После щелчка на кнопке **Finish** (на промежуточном этапе нужно в информационном окне щелкнуть кнопку **ОК**) получаем новый проект. Окно нового проекта представлено на рис. 13.4.

Особенность используемого в Microsoft Visual C++ компилятора состоит в том, что в шапке заголовков программы необходимо указать инструкцию `#include "stdafx.h"`. Во всем остальном ничего необычного: в окне справа вводится программный код, после чего щелкают пиктограмму с изо-

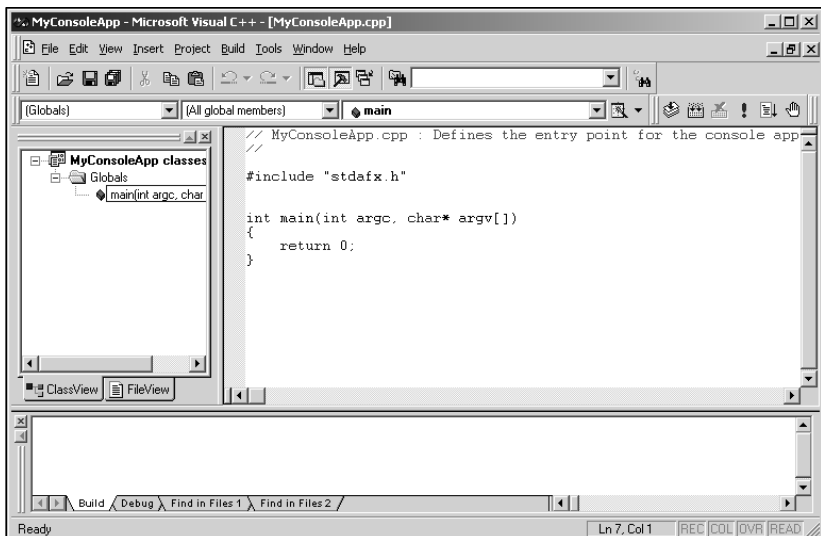


Рис. 13.4. Окно приложения Microsoft Visual C++ с файлами проекта

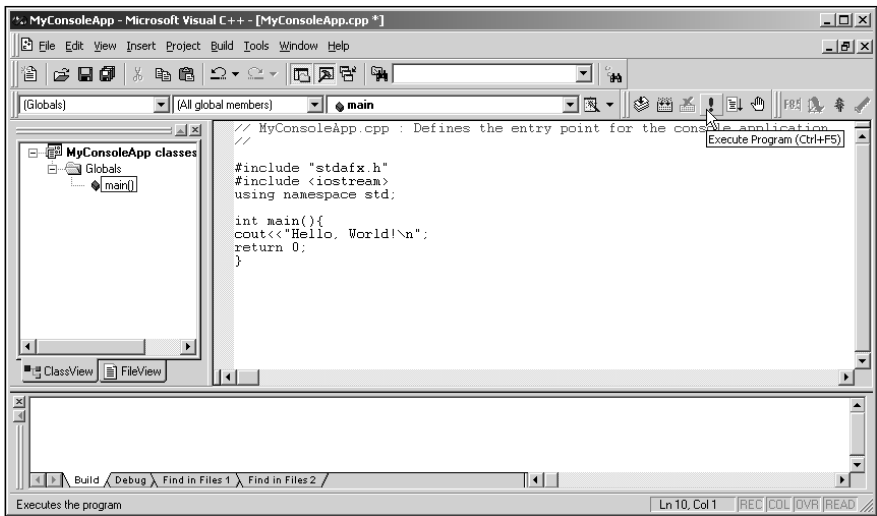


Рис. 13.5. Запуск программы на выполнение

бражением восклицательного знака или используют соответствующую команду **Execute** из меню **Build** (рис. 13.5).

При этом программа автоматически компилируется и, если компиляция прошла успешно, сразу запускается на выполнение. Если при компиляции возникают ошибки, их описание отображается в нижней части (в нижнем окне отладки) главного окна приложения Microsoft Visual C++. Результат выполнения программы отображается в окне консоли, как показано на рис. 13.6.

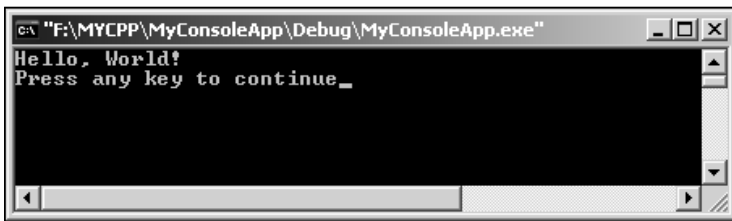


Рис. 6. Результат выполнения программы

Сообщение **Press any key to continue** добавляется автоматически. Чтобы оно отображалось в новой строке, а результат выполнения программы (выводимый текст) выглядел эстетично, обычно в конце после вывода программой текста добавляется инструкция перехода к новой строке.

С точки зрения выполнения набора программного кода, его компиляции и выполнения в среде Microsoft Visual C++ Express Edition, по сравнению с предыдущим случаем, отличия не очень большие. Окно приложения Microsoft Visual C++ Express Edition представлено на рис. 13.7.

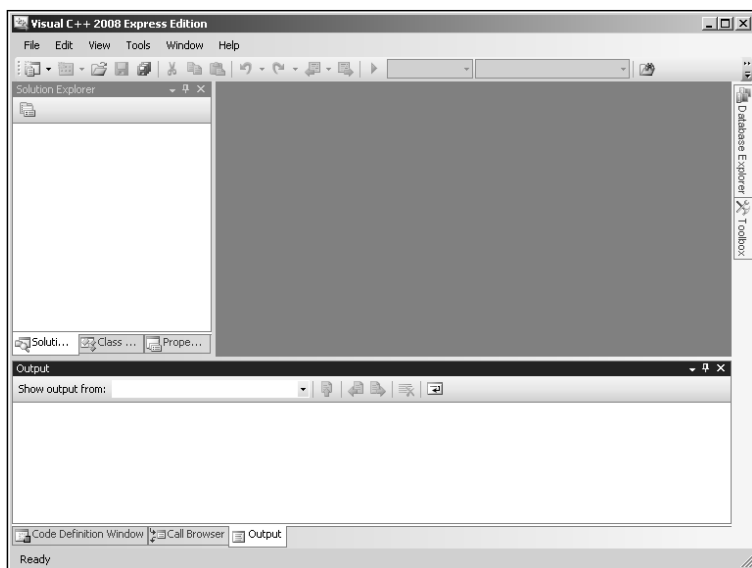


Рис. 13.7. Окно приложения *Microsoft Visual C++ Express Edition*

Для создания нового проекта необходимо выбрать команду **File ► New ► Project**, после чего открывается окно **New Project** для выбора типа создаваемого приложения, представленное на рис. 13.8.

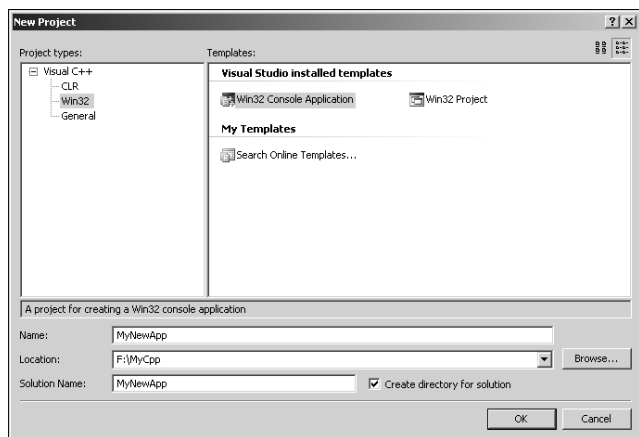


Рис. 13.8. Выбор типа создаваемого приложения, его имени и места сохранения

После подтверждения сделанных настроек (щелчок на кнопке **OK**) последовательно появляется еще ряд окон, наподобие того, что представлено на рис. 13.9, в которых необходимо согласиться с предлагаемыми настройками.

В поле слева выбирается тип приложения (**Win32** в данном случае), а в поле справа необходимо указать шаблон (консольное приложение – это **Win32 Console Application**). Внизу в поле **Name** указывается имя проекта (программы), а в поле **Location** необходимо указать полный путь к папке, в которой будет сохранен проект.

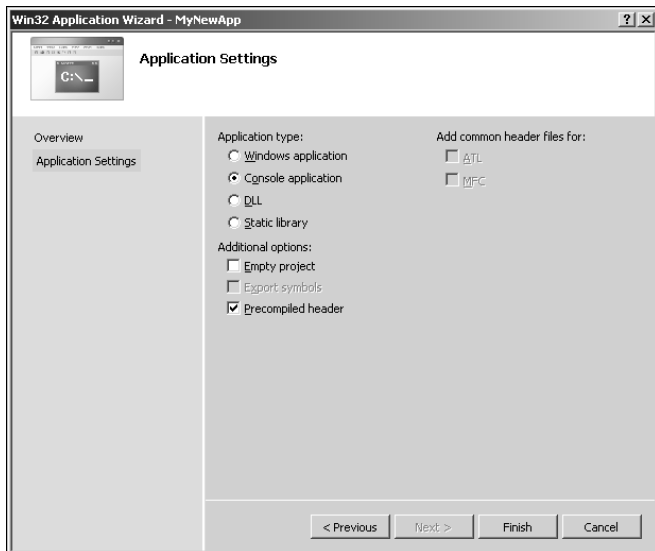


Рис. 13.9. Параметры создаваемого консольного приложения

После щелчка на кнопке **Finish** в редакторе приложения Microsoft Visual C++ Express Edition появятся файлы нового проекта (рис. 13.10).

Как и в случае с Visual C++, при работе со средой Microsoft Visual C++ Express Edition в заголовке программы необходимо добавить инструкцию `#include "stdafx.h"`.

Кроме того, в данном случае перед окончанием про-

граммы имеет смысл разместить команду `system("PAUSE")` для того, чтобы консольное окно не убиралось автоматически с экрана сразу по завершении работы программы. Если программа выполняется достаточ-

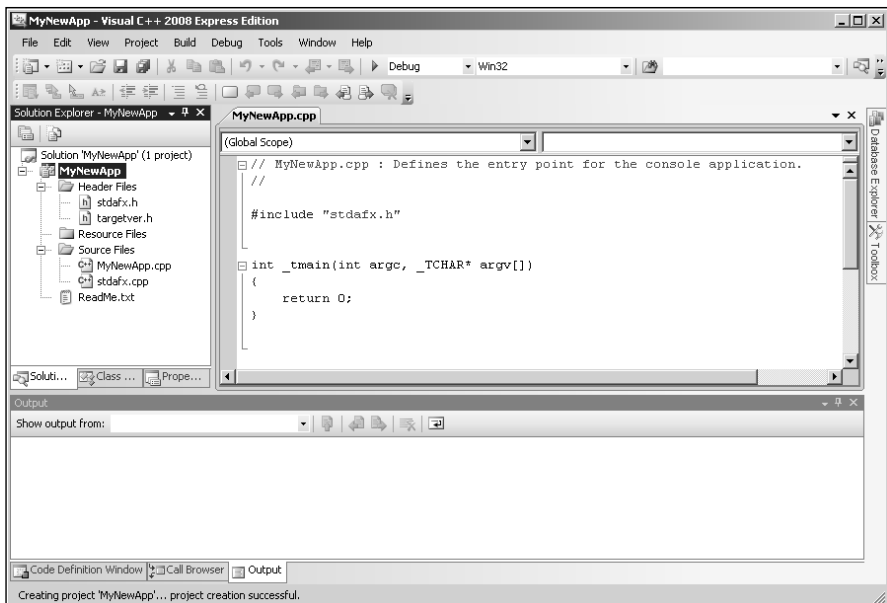


Рис. 13.10. Окно приложения Microsoft Visual C++ Express Edition с файлами проекта

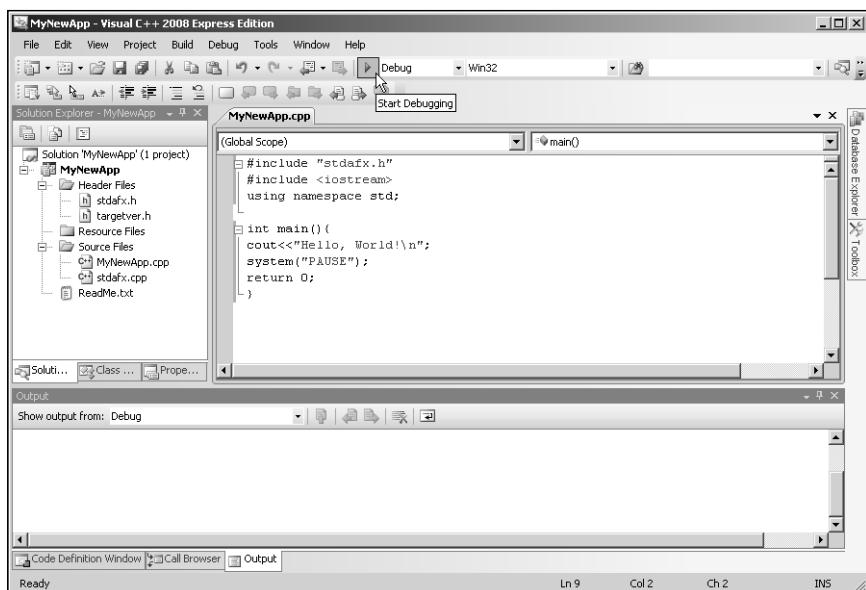


Рис. 13.11. Окно приложения Microsoft Visual C++ Express Edition с программным кодом, готовым к компиляции и выполнению

но быстро, можно вообще не заметить, что консольное окно появлялось на экране. Программный код в редакторе Microsoft Visual C++ Express Edition с инструкцией `system("PAUSE")` представлен на рис. 13.11.

Для компиляции и выполнения программы щелкают кнопку с изображением треугольной стрелки (см. рис. 13.11) или выбирают команду **Debug ▶ Start Debugging**. Результат выполнения программы показан на рис. 13.12.

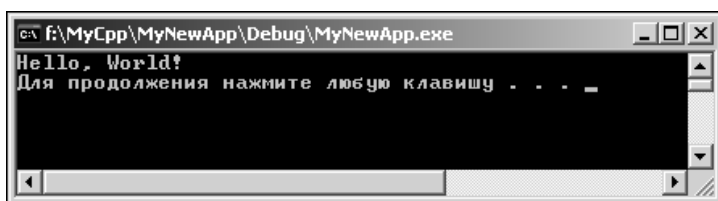


Рис. 13.12. Результат выполнения программы

Пожалуй, самое большое удобство работы с продуктами компании Microsoft состоит в том, что предлагаемые к услугам пользователей и программистов утилиты позволяют решать практически любые задачи, при этом установка и использование соответствующих программных продуктов особых проблем не вызывает.

Достаточно интересными являются проекты Eclipse и NetBeans, хотя продуктивное использование этих свободно распространяемых продуктов тре-

бует некоторых познаний в области настройки компиляторов и оболочек. Базовым языком обоих продуктов является Java, и для их работы, как отмечалось, необходимо установить и настроить виртуальную Java-машину. Кроме того, отдельно устанавливаются компилятор и выполняются настройки. Для загрузки и установки компиляторов и сопутствующих утилит можно порекомендовать обратиться к сайтам www.mingw.org или www.cygwin.com. Что касается непосредственно установки утилит Eclipse и NetBeans, необходимо также учесть наличие различных версий и дистрибутивов – выбирать нужно те, что поддерживают работу с языками C/C++. На рис. 13.13 представлено окно приложения NetBeans Pack C/C++, поддерживающего работу с кодами C++.

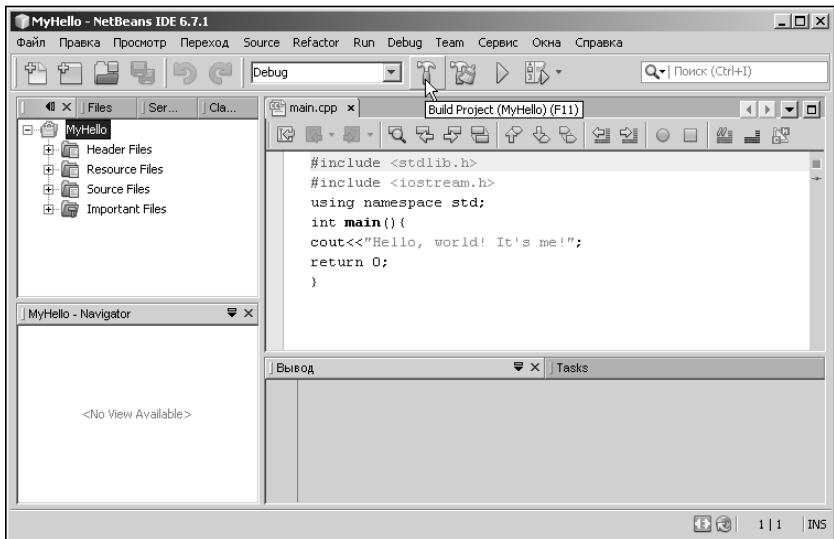


Рис. 13.13. Рабочее окно утилиты NetBeans Pack C/C++

Набранный в редакторе программный код компилируется и запускается на выполнение – для этого имеются специальные кнопки на панели инструментов и команды меню. К достоинствам пакета NetBeans Pack C/C++ можно отнести то, что он имеет русифицированную версию – для новичков это облегчает работу. Идеологически близким к пакету NetBeans представляется и пакет Eclipse. Окно утилиты Eclipse представлено на рис. 13.14.

Для создания нового проекта в Eclipse выбираем команду **File ► New ► C++ Project**, в результате чего открывается окно создания проекта **C++ Project** (рис. 13.15).

Самый простой способ создать рабочее приложение на C++ – в левом поле **Project type** выбрать **Hello World C++ Project**. В поле **Project name** указывается имя проекта, а в поле **Location** указывается папка для сохране-

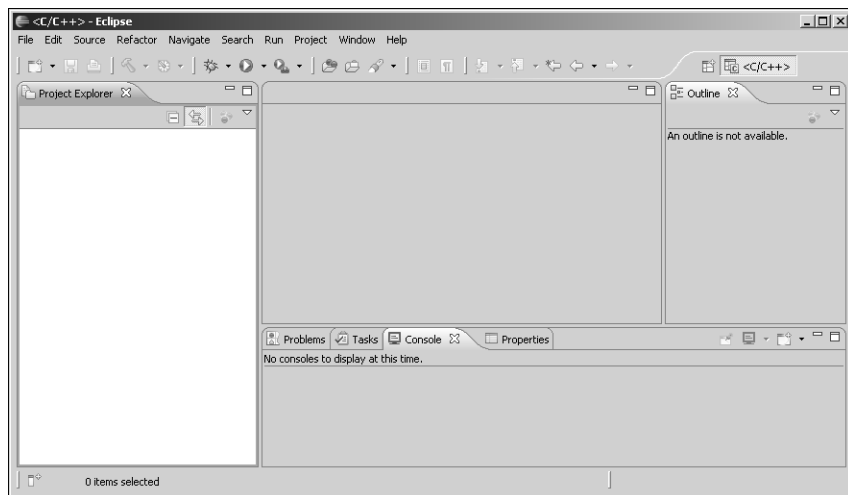


Рис. 13.14. Окно утилиты Eclipse

ния проекта. После щелчка на кнопке **Finish** получаем шаблонное приложение, которое путем редактирования программного кода доводим до приемлемого вида, как, например, на рис. 13.16.

После того, как код набран, его необходимо откомпилировать, для чего используют команду **Project ► Build Project**. Для запуска программы используют команду **Run ► Run** (или соответствующую кнопку на панели инструментов главного окна приложения Eclipse). Результат выполнения консольной программы можно наблюдать на вкладке **Console** в нижней части главного окна приложения Eclipse, как показано на рис. 13.17.

К достоинствам приложения Eclipse следует отнести возможность использования для вывода в консоль русского текста (пример представлен на рис. 13.18).

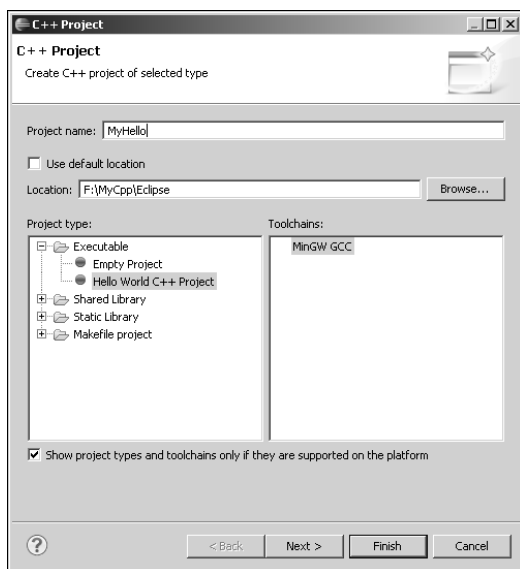


Рис. 13.15. Окно создания проекта

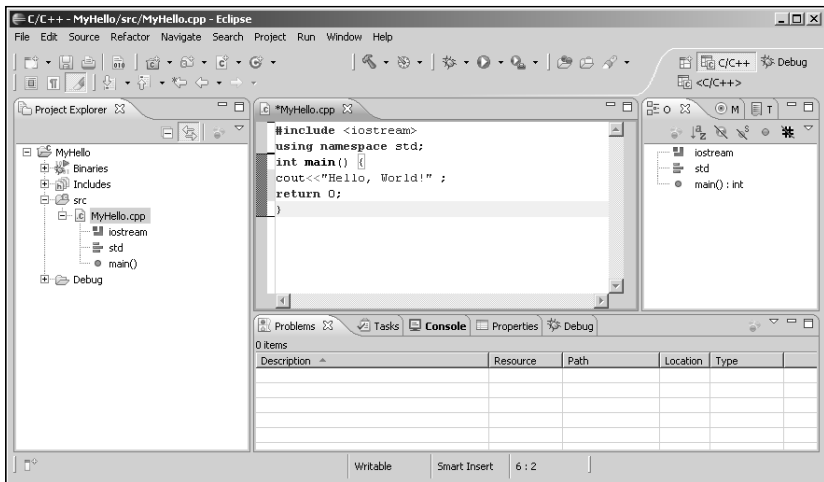


Рис. 13.16. Окно редактора с программным кодом

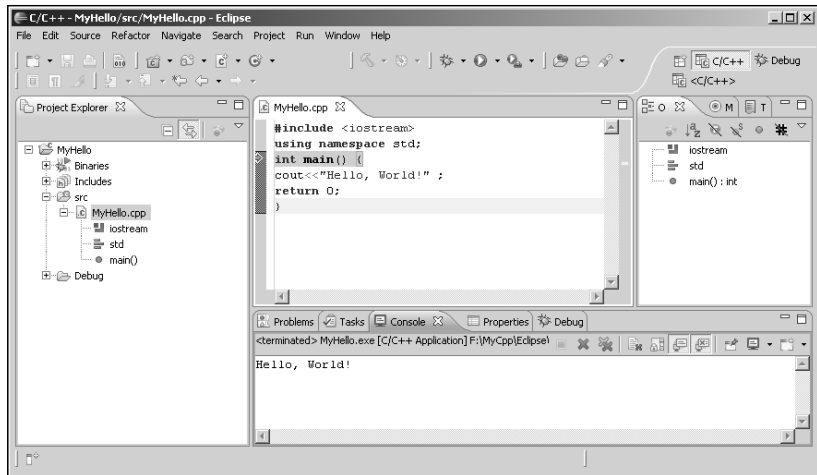


Рис. 13.17. Результат выполнения программы

Тем не менее, работа с этим приложением требует некоторых навыков и усердия – но оно того стоит!

Достаточно неплохо зарекомендовала себя и бесплатная среда разработки Dev-C++. К несомненным преимуществам следует отнести также доступность, легкость установки и использования, наличие русифицированной версии. Эта среда разработки более детально описывается в следующем приложении.

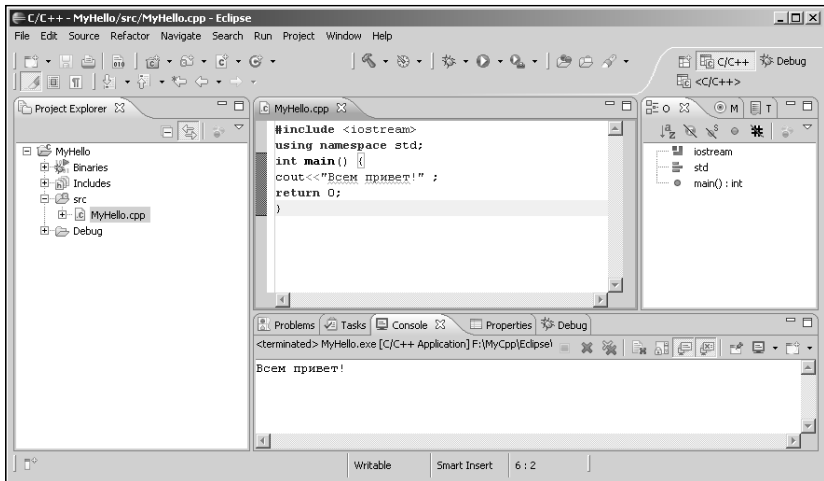


Рис. 13.18. Вывод программой русского текста

Глава 14.

Работа в Dev-C++

Для того чтобы продать что-нибудь ненужное, нужно сначала купить что-нибудь ненужное. А у нас денег нет!

из м/ф "Трое из Простоквашино"

Интегрированная среда разработки Dev-C++ является простым, надежным и эффективным средством для написания программ на C++. Все, что необходимо для работы с Dev-C++ можно загрузить и установить абсолютно легально и бесплатно с сайта www.bloodshed.org. Весь процесс установки достаточно прост и понятен, так что особых комментариев не требует. В частности, при установке выбирается язык интерфейса – приложение Dev-C++ русифицировано. Язык интерфейса можно изменить и после установки. Важно и то, что автоматически устанавливаются компилятор и отладчик (в Dev-C++ используется компилятор gcc и отладчик gdb), но пользователю об этом знать и не обязательно. Более того, все необходимые настройки также выполняются автоматически. Так что сразу после установки приложение Dev-C++ готово к использованию. На рис. 14.1 показано окно приложения Dev-C++.

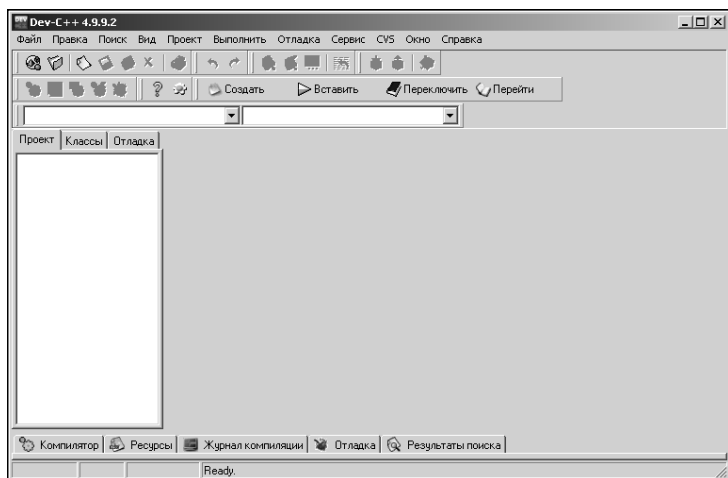


Рис. 14.1. Окно приложения Dev-C++

Для создания консольного проекта с помощью приложения Dev-C++ в меню **Файл** выбираем команду **Проект** подменю **Создать**. В результате открывается окно **Новый проект**, представленное на рис. 14.2.

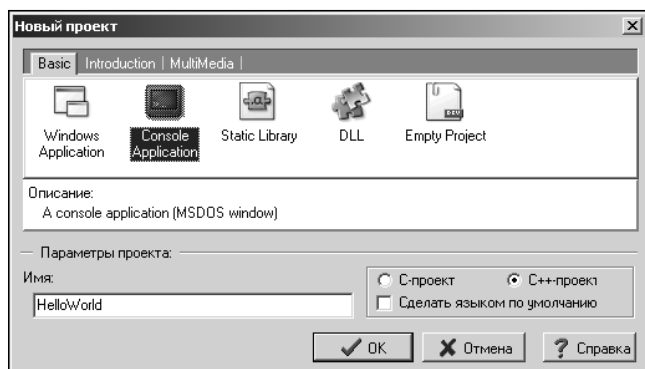


Рис. 14.2. Окно создания нового проекта Новый проект открыто на вкладке Basic

Окно содержит несколько вкладок (**Basic**, **Introduction** и **Multimedia** – содержат шаблоны для различных приложений). На вкладке Basic выбирается тип приложения (для консольного приложения это **Console Application**), язык (C или C++) и в поле **Имя** указывается имя проекта. После щелчка на кнопке **ОК** предлагается выбрать папку для сохранения проекта. По умолчанию предлагается папка, указанная как рабочая папка для приложения Dev-C++ (способ выполнения этой настройки описывается далее). Создастся новый проект, а окно редактора приложения Dev-C++ становится таким, как на рис. 14.3.

Все, что осталось, – внести правку в предлагаемый по умолчанию шаблон для программного кода, откомпилировать код и запустить программу на выполнение. Обращаем внимание, что в Dev-C++ автоматически предлага-

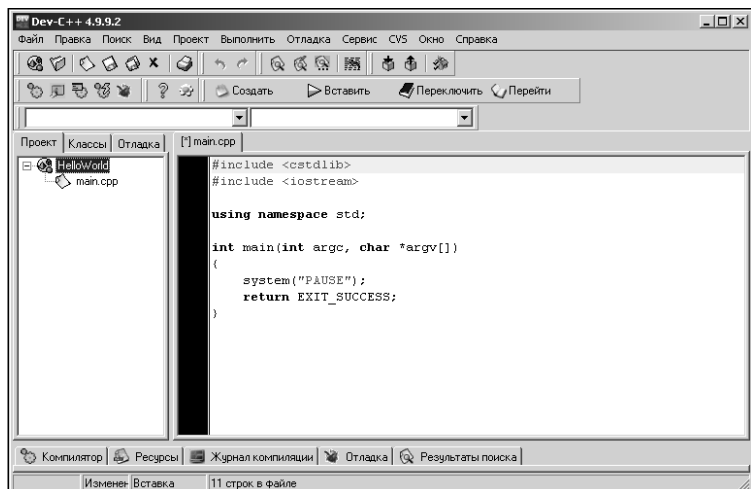


Рис. 14.3. Окно приложения Dev-C++ с колом нового проекта

ется использовать инструкцию `system("PAUSE")` для задержки консольного окна на экране. Все остальное – как обычно. Для компилирования и запуска программы можно воспользоваться командой **Выполнить ► Скомпилировать и выполнить**, но лучше щелкнуть на соответствующей кнопке на панели инструментов (рис. 14.4).

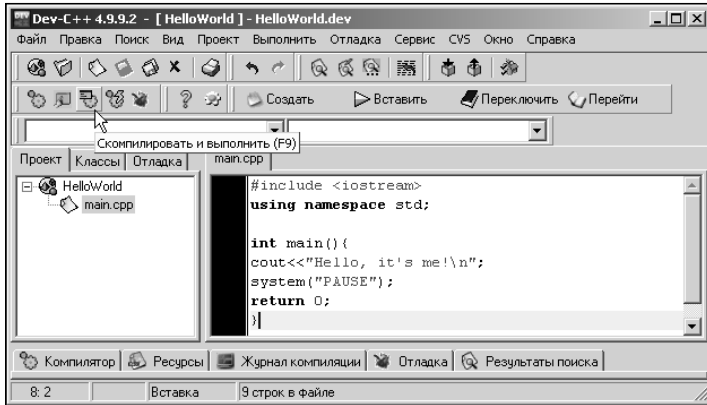


Рис. 14.4. Компилирование и запуск программы на выполнение

В результате программа компилируется, и, если все прошло успешно, запускается на выполнение. В данном случае результат такой, как показано на рис. 14.5.

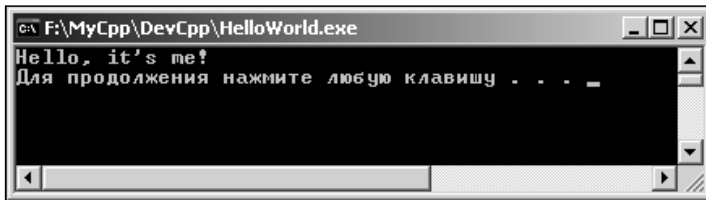


Рис. 14.5. Результат выполнения программы

Строго говоря, выше приведен тот минимальный объем информации, который необходим для написания программ с помощью среды разработки Dev-C++. Разумеется, это далеко не все возможности, предоставляемые пользователю при работе с данным приложением. Некоторые, наиболее полезные и интересные, рассмотрим подробнее.

В первую очередь дадим общий обзор элементов интерфейса приложения Dev-C++. Основные его элементы – это панель меню, несколько панелей инструментов с кнопками, дублирующими команды меню, и ряд управляющих элементов в виде вкладок и раскрывающихся списков.

Панели инструментов, расположенные сразу под панелью меню, содержат кнопки для выполнения стандартных действий: создание нового проекта,

открытие нового проекта или файла, вставка нового файла, сохранение, закрытие проекта, вывод на печать и прочее. Хотя может показаться, что панель инструментов под панелью меню одна, на самом деле их несколько. Это же относится и ко "второму эшелону" панелей инструментов: там расположены кнопки для компилирования, выполнения, компоновки и отладки программного кода. Панели можно удалять или добавлять с помощью контекстного меню, установив или убрав соответствующий значок (рис. 14.6).

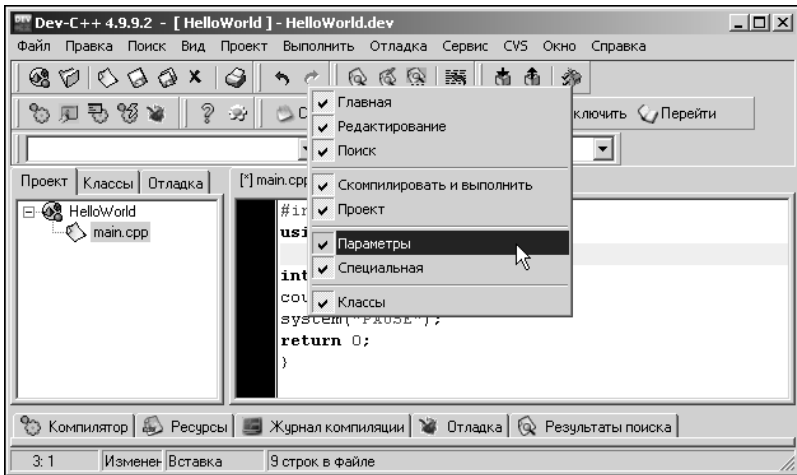


Рис. 14.6. Использование контекстного меню для отображения и скртия панелей инструментов

Каждую кнопку в отдельности описывать не будем в силу сразу нескольких

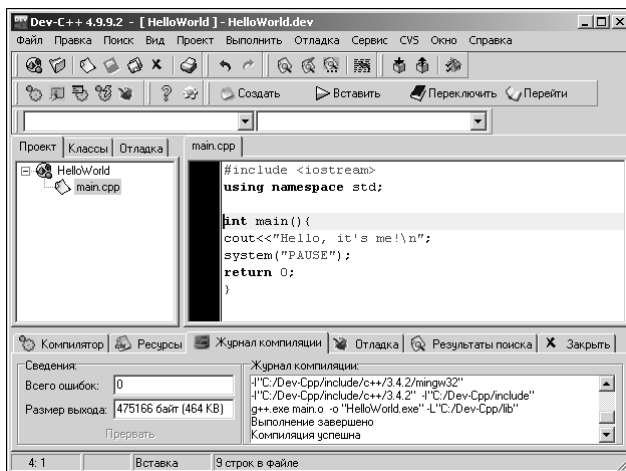


Рис. 14.7. Открыто подокно с отчетом о процессе компиляции

причин. Во-первых, в Dev-C++ можно менять внешний стиль интерфейса приложения, в силу чего изображения и вид кнопок меняются. Во-вторых, кнопок достаточно много, и их детальное описание совершенно не вписывается в идеологию книги. В-третьих, и это самое главное, при наведении курсора мыши на кнопку отображает-

ся подсказка, которая обычно в полной мере дает представление о том, для чего нужна та или иная кнопка. Эта же подсказка содержит горячее сочетание клавиш.

Управляющие элементы в виде вкладок используются для переключения между подокнами или отображения подокон. Например, если щелкнуть вкладку **Журнал компиляции**, в нижней части окна приложения Dev-C++ появится подокно с отчетом о процессе компиляции программы. Убрать подокно можно щелчком на вкладке **Заккрыть**.

Что касается панели меню, то оно содержит несколько пунктов, кратко описанных в табл. 14.1.

Табл. 14.1. Панель меню приложения Dev-C++

Меню	Назначение
Файл	Содержит команды создания нового проекта, закрытия, сохранения, импорта, экспорта, вывода на печать и завершения работы с приложением
Правка	Содержит команды по редактированию кода. В частности, отмены и повторения отмененного действия, копирования, вставки и удаления в буфер, добавления закладок и комментариев и прочее
Поиск	Содержит команды для выполнения поиска в программном коде в различных режимах
Вид	Содержит команды настройки внешнего вида окна приложения, такие как отображение строки состояния, панелей инструментов и сообщений компилятора
Проект	Содержит несколько важных команд, как то: создание и добавление файла к проекту, удаление из проекта и команду настройки параметров проекта
Выполнить	Содержит команды для компилирования и выполнения программы
Отладка	Меню содержит команды, используемые при отладке программы
Сервис	Очень полезное меню при выполнении и изменении настроек компилятора, редактора и среды, настройке комбинации клавиш и прочих параметров
CVS	Меню для работы с утилитами CVS
Окно	Меню содержит команды для управления окнами
Справка	Справочная система

На некоторых настройках остановимся подробнее. Так, воспользовавшись командой **Сервис ► Параметры среды**, увидим окно **Параметры среды**, содержащее несколько вкладок (рис. 14.8).

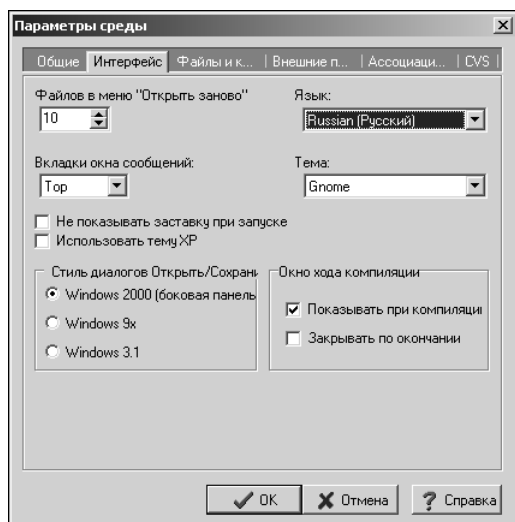


Рис. 14.8. Окно *Параметры среды* открыто на вкладке *Интерфейс*

На вкладке **Интерфейс** можно выполнить ряд базовых настроек, и в первую очередь задать язык (раскрывающийся список **Язык**) интерфейса и стиль окна приложения (раскрывающийся список **Тема**). На рис. 14.9 показан результат применения темы **New Look** (до этого использовалась тема **Gnome**) и английского языка в качестве базового языка интерфейса приложения Dev-C++.

На вкладке **Файлы и каталоги** диалогового окна **Параметры среды** (рис. 14.10) задаются

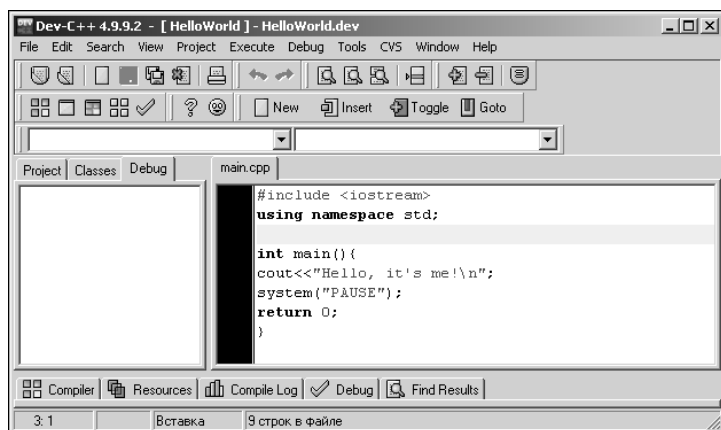


Рис. 14.9. Тема *New Look* и английский язык интерфейса

каталоги для хранения файлов, создаваемых или используемых при работе приложения.

Стоит обратить внимание на поле **Каталог пользователя**, в котором указывается каталог, предлагаемый по умолчанию при сохранении файлов нового проекта.

Достаточно широкие возможности в Dev-C++ имеются по настройке редактора программных кодов. Настройки выполняются в окне **Параметры редактора**, которое открывается посредством команды **Сервис ► Параметры редактора** (рис. 14.11).

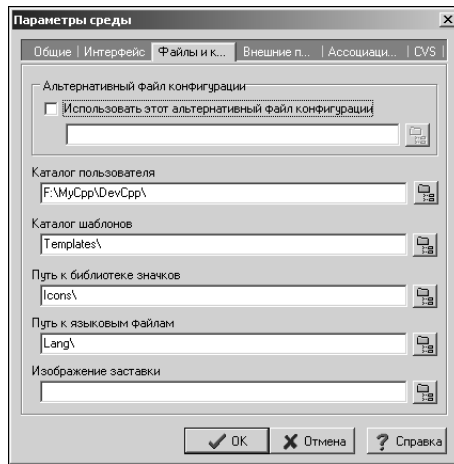


Рис. 14.10. Окно Параметры среды открыто на вкладке Файлы и каталоги

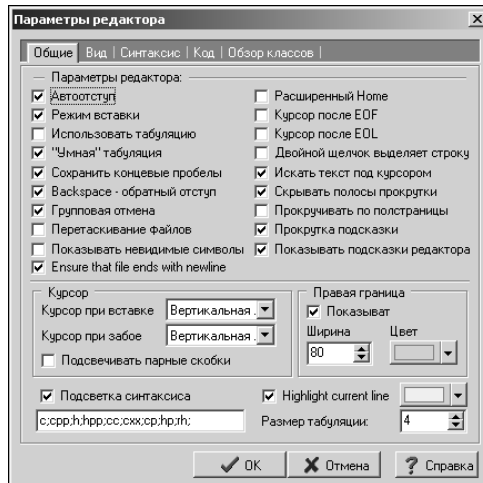


Рис. 14.11. Окно Параметры редактора открыто на вкладке Общие

Окно содержит несколько вкладок и позволяет задавать самые различные параметры и режимы редактора, включая шрифт и способ выделения синтаксических конструкций. Правда, здесь хочется отметить, что вносить изменения в настройки по умолчанию имеет смысл только в том случае, когда пользователь уверен в своих действиях. Особенно это актуально для настроек компилятора, которые выполняются в окне **Параметры компилятора** (открывается с помощью одноименной команды) – менять настройки компилятора можно только тогда, когда не менять их нельзя.

■ Благодарности

От автора. В первую очередь я благодарен доктору физ.-мат. наук, профессору, академику НАН Украины Булавину Леониду Анатольевичу, который, будучи деканом физического факультета Киевского национального университета имени Тараса Шевченко, привлек меня к работе с компьютерами и программным обеспечением на факультете и создал для этого все условия, а также нынешнему декану факультета, доктору физ.-мат. наук, профессору Макарку Николаю Владимировичу, который продолжает эту приятную для меня традицию. Я искренне рад, что имею возможность читать курс объектно-ориентированного программирования для студентов медико-инженерного факультета национального технического университета «Киевский Политехнический Институт», и искренне благодарю за предоставленную возможность декана этого факультета, доктора медицинских наук, профессора Яценко Валентина Порфирьевича, а также заведующего кафедрой биомедицинской инженерии, доктора медицинских наук, профессора Максименко Виталия Борисовича.

Возможность читать лекции по C++ появилась не без активного участия заведующего кафедрой теоретической физики физического факультета Киевского национального университета имени Тараса Шевченко, доктора физ.-мат. наук, профессора Ежова Станислава Николаевича, за что ему спасибо. Наконец, (last but not the least) я выражаю искреннюю благодарность человеку, который учил меня программированию, а теперь моему коллеге (что вдвойне приятно), кандидату физ.-мат. наук, доценту кафедры теоретической физики Усенко Константину Владимировичу.

От редакции. Редакция выражает благодарность Арно В. Г. за помощь в работе над книгой и ценные консультации.

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *М. А. Финкова*

Корректоры: *А. В. Громова*

ООО «Наука и Техника»

Лицензия №000350 от 23 декабря 1999 года.

198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.

Подписано в печать 18.02.2016. Формат 70х100 1/16.

Бумага газетная. Печать офсетная. Объем 30 п. л.

Тираж Заказ