

## Н. А. Литвиненко





Допущено УМО вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению 654600

«Информатика и вычислительная техника»

Санкт-Петербург

«БХВ-Петербург» 2005

УДК 681.3.068+800.92С++(075.8) ББК 32.973.26-018.1я73

Л64

**Литвиненко Н. А.**

Л64 Технология программирования на С++. Начальный курс. — СПб.: БХВ-Петербург, 2005. — 288 с.: ил.

ISBN 5-94157-655-2

Рассмотрены основы программирования на С++, начиная с описания синтаксиса языка С, приемов и методов программирования в стиле класси- ческого С до введения понятий классов, шаблонов классов и вопросов наследования. Уделено особое внимание использованию стандартной биб- лиотеки шаблонов STL. Представлен современный подход к организации ввода/вывода при помощи потоковых классов. Рассматривается техника создания простейших Windows-приложений с использованием API-функ- ций. Материал иллюстрируется многочисленными примерами.

*Для студентов и преподавателей технических вузов и самообразования*

УДК 681.3.068+800.92С++(075.8) ББК 32.973.26-018.1я73

**Группа подготовки издания:**

Главный редактор *Екатерина Кондукова*

Зам. главного редактора *Игорь Шишигин*

Зав. редакцией *Григорий Добин*

Редактор *Дарья Масленникова* Компьютерная верстка *Ольги Сергиенко* Корректор *Зинаида Дмитриева*

Дизайн обложки *Игоря Цырульникова*

Зав. производством *Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 22.04.05.

Формат 70 1001/16. Печать офсетная. Усл. печ. л. 23,22.

Тираж 3000 экз. Заказ №

"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию

№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов в ГУП "Типография "Наука"

199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-655-2 © Литвиненко Н. А., 2005

© Оформление, издательство "БХВ-Петербург", 2005

# Оглавление

[Введение 7](#_TOC_250104)

Глава 1. Основы программирования на С++ 9

[Структура классической С-программы 9](#_TOC_250103)

[Препроцессор 10](#_TOC_250102)

[Создание простой программы 11](#_TOC_250101)

[Типы данных. Описание переменных 15](#_TOC_250100)

Переменные целого типа: bool, char, short, int, long 18

*enum* — перечисляемый тип 18

Переменные с плавающей точкой: float, double, long double 19

[Целые константы 20](#_TOC_250099)

[Константы с плавающей точкой 21](#_TOC_250098)

[Символьные константы 21](#_TOC_250097)

[Строковые константы 22](#_TOC_250096)

[Переменные-константы 23](#_TOC_250095)

[Комментарии 24](#_TOC_250094)

[Классы памяти 24](#_TOC_250093)

[Операторы и операции 25](#_TOC_250092)

[Арифметические операции 25](#_TOC_250091)

[Приведение типов 26](#_TOC_250090)

Операции ++ и -- 27

[Битовые операции 27](#_TOC_250089)

[Комбинированные операции 28](#_TOC_250088)

[Операции отношения 29](#_TOC_250087)

[Логические операторы 29](#_TOC_250086)

[Операторы языка С++ 30](#_TOC_250085)

[Оператор перехода *goto* 30](#_TOC_250084)

Условный оператор if . . . else 31

[Условный арифметический оператор 31](#_TOC_250083)

[Операторы цикла 31](#_TOC_250082)

[Простая программа с циклами 33](#_TOC_250081)

Вспомогательные операторы: break, continue 34

Переключатель switch 34

[Массивы и указатели 35](#_TOC_250080)

[Примеры программ с использованием массивов 36](#_TOC_250079)

[Указатели 38](#_TOC_250078)

[Определение псевдонимов 41](#_TOC_250077)

[Многомерные массивы 41](#_TOC_250076)

[Динамическое выделение памяти 42](#_TOC_250075)

[Функции 49](#_TOC_250074)

[Встраиваемые функции 51](#_TOC_250073)

4 Оглавление

[Указатели на функции 52](#_TOC_250072)

[Примеры некоторых функций работы со строками 52](#_TOC_250071)

[Ссылки 54](#_TOC_250070)

[Примеры простых вычислительных задач 55](#_TOC_250069)

[Структуры 61](#_TOC_250068)

[Объединения 63](#_TOC_250067)

[Описание головного модуля 65](#_TOC_250066)

[Вопросы к главе 67](#_TOC_250065)

[Задание для самостоятельной работы 68](#_TOC_250064)

Глава 2. Классы 70

[Создание нового класса в среде Visual C++ 6.0 71](#_TOC_250063)

[Подробное описание класса *Time* 73](#_TOC_250062)

Класс String 79

[Пространство имен 87](#_TOC_250061)

[Наследование 89](#_TOC_250060)

[Наследование на примере классов: Работник -> Менеджер -> Ученый 89](#_TOC_250059)

Использование классов, как пользовательских типов данных 93

[Виртуальные методы и классы 96](#_TOC_250058)

[Классы, объявленные как виртуальные 99](#_TOC_250057)

[Шаблоны 100](#_TOC_250056)

[Шаблон функции 100](#_TOC_250055)

[Шаблон класса 102](#_TOC_250054)

[Обработка исключений 107](#_TOC_250053)

[Вопросы к главе 112](#_TOC_250052)

[Задание для самостоятельной работы 113](#_TOC_250051)

Глава 3. Потоки 115

[Консольный ввод/вывод 115](#_TOC_250050)

[Флаги 116](#_TOC_250049)

[Манипуляторы 117](#_TOC_250048)

[Методы 118](#_TOC_250047)

[Память как поток 122](#_TOC_250046)

[Файловый ввод/вывод 125](#_TOC_250045)

[Произвольный доступ к файлам 129](#_TOC_250044)

[Доступ к файловому буферу 131](#_TOC_250043)

[Итераторы потоковых буферов 132](#_TOC_250042)

[Вопросы к главе 135](#_TOC_250041)

[Задание для самостоятельной работы 135](#_TOC_250040)

Глава 4. Стандартная библиотека шаблонов STL 137

Контейнер Vector 138

[Операции с векторами 139](#_TOC_250039)

[Алгоритмы 145](#_TOC_250038)

[Алгоритмы поиска 146](#_TOC_250037)

[Модифицирующие алгоритмы 157](#_TOC_250036)

[Алгоритмы упорядоченных интервалов 173](#_TOC_250035)

[Численные алгоритмы 179](#_TOC_250034)

Оглавление 5

Контейнер Deque 183

[Операции с деками 183](#_TOC_250033)

Контейнер List 187

Контейнеры Set, Multiset 191

Контейнеры Map, Multimap 195

[Итераторы 200](#_TOC_250032)

[Итераторы ввода 200](#_TOC_250031)

[Итераторы вывода 200](#_TOC_250030)

[Прямые итераторы 201](#_TOC_250029)

[Двунаправленные итераторы 202](#_TOC_250028)

[Итераторы произвольного доступа 203](#_TOC_250027)

[Обратные итераторы 203](#_TOC_250026)

[Итераторы вставки 205](#_TOC_250025)

[Вспомогательные функции итераторов 207](#_TOC_250024)

[Специальные контейнеры 208](#_TOC_250023)

Контейнер Stack 208

Контейнер Deque 209

Контейнер Priority\_queue 210

Класс string 211

[Функции ввода/вывода 220](#_TOC_250022)

[Заключение 222](#_TOC_250021)

[Вопросы к главе 222](#_TOC_250020)

[Задание для самостоятельной работы 223](#_TOC_250019)

Глава 5. Организация оконного интерфейса 224

[Каркас Windows-приложения 226](#_TOC_250018)

[Исследование программы 230](#_TOC_250017)

[Обработка сообщений 235](#_TOC_250016)

[Нажатие клавиши 236](#_TOC_250015)

[Сообщение мыши 240](#_TOC_250014)

[Создание окна 242](#_TOC_250013)

[Таймер 242](#_TOC_250012)

[Рисование в окне 245](#_TOC_250011)

[Работа с текстом 262](#_TOC_250010)

[Диалог с пользователем 270](#_TOC_250009)

[Окно сообщений 270](#_TOC_250008)

[Меню 271](#_TOC_250007)

[Заключение 278](#_TOC_250006)

[Вопросы к главе 278](#_TOC_250005)

[Задание для самостоятельной работы 279](#_TOC_250004)

[Приложение 281](#_TOC_250003)

[Рекомендуемая литература 283](#_TOC_250002)

[Дополнительная литература 283](#_TOC_250001)

[Предметный указатель 284](#_TOC_250000)

# Введение

Основной причиной, побудившей автора взяться за написание данной книги, послужило желание создать учебник для студентов, изучающих дисциплины "Технология программирования" и "Объектно-ориентированное программи- рование". Несмотря на обилие различной литературы по языку программиро- вания С++, сложно порекомендовать какую-то одну книгу в качестве подоб- ного учебника. Из недавно изданных можно отметить учебник и практикум Павловской [3, 4] — серьезные издания со строгим изложением материала, хорошо подходящие в качестве справочных пособий. Однако для первона- чального изучения языка требуется менее формализованный подход, который автор и попытался реализовать в данном учебном пособии, построенном на основе курса лекций, читаемых студентам специальностей "Программное обеспечение вычислительной техники и автоматизированных систем" и "Ин- формационные системы и технологии". В качестве дополнительной литера- туры можно рекомендовать очень неплохое издание Р. Лафоре [2] из серии "Классика Computer Science". Книга отличается хорошим стилем изложения, но приведенные в ней примеры не всегда удачны.

При построении курса возникает вопрос о его структуре и содержании. Язык программирования С++ является языком объектно-ориентированного про- граммирования (ООП). Существует несколько подходов к изучению данного языка. Иногда начинают с классического С и лишь после этого переходят к расширению языка С++, собственно классам и ООП. В последнее время мно- гие авторы сразу начинают изложение с введения классов, однако автор дан- ной книги не считает это правильным, выбирая гибридный подход к изложе- нию материала — расширения языка включены с самого начала описания, а переход к классам осуществлен лишь после того, как детально разобран про- цедурный подход к программированию.

Правда, и здесь трудно удержаться и не воспользоваться возможностями по- токового ввода/вывода. Этот современный подход используется в данной книге сразу, а соответствующие функции классического С не рассмотрены.

*8 Введение*

В первых главах книги не рассмотрены также и графические построения, по- скольку в современных операционных системах графика тесно связана с оконным интерфейсом, чему у нас посвящена отдельная, последняя глава. В ней собраны простейшие графические функции и приведены примеры их использования.

Книга состоит из пяти глав.

*Глава 1. "Основы программирования на С++"* — описание синтаксиса язы- ка С, рассмотрение приемов и методов программирования в стиле класси- ческого С.

*Глава 2. "Классы"* — вводится понятие класса, рассматриваются шаблонные классы, вопросы наследования и построения сложных классов. Приводятся многочисленные примеры.

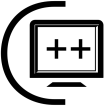
*Глава 3. "Потоки"* — рассматривается современный подход к организации ввода/вывода при помощи потоковых классов.

*Глава 4. "Стандартная библиотека шаблонов STL"* — описание стандартной библиотеки. На многочисленных примерах рассмотрено использование STL.

*Глава 5. "Организация оконного интерфейса"* — рассматривается структура скелета Windows-программы, методы обработки стандартных сообщений. Приведены многочисленные примеры вывода текста и простейшей графики. Программирование основано на непосредственном использовании API- функций.

Сразу оговоримся для будущих критиков, что данная книга ни в коей мере не претендует ни на полноту, ни на строгость, все это принесено в жертву ясно- сти изложения. Некоторые вопросы рассматриваются поверхностно, некото- рые вообще опущены. Это лишь вводный курс, цель которого — изучение синтаксиса языка С++ и основных методов программирования. В книге дос- таточно поверхностно рассмотрено создание Win32-приложений — лишь на том уровне, чтобы у читателя сложилось представление о структуре стан- дартной Windows-программы и принципах ее функционирования.

В заключение отметим, что все примеры, рассмотренные в книге, являются фрагментами работающих программ, а тексты некоторых программ при- ведены полностью в листингах. Все программы тестировались в MS Visual C++ 6.0 под управлением операционной системы Windows XP.

**Г ЛА В А 1**

# Основы программирования на С++

Часто формализм языка рассматривается безотносительно конкретной реали- зации. И все же читателю будет предложено использовать в качестве основ- ного инструмента для построения программ консольное приложение среды MS Visual C++ 6.0 (Win32 Console Application), ориентируясь на приведенные далее примеры, реализованные в этой среде программирования. Язык про- граммирования в классическом стиле, рассмотренный в данной главе, будет дополнен лишь элементами расширения, принципиально не меняющими его сути. Однако для организации ввода/вывода воспользуемся подходом С++, применив потоковые классы, подробнее о которых будет рассказано далее, в *главе 3*.

## Структура классической С-программы

|  |  |
| --- | --- |
| Объявление файлов  #*Директивы препроцессора* включения | |
|  | |
| *Объявление функций* | Описание прототипов  функций |
| *Описание глобальных переменных и констант* |  |
| main*(параметры)* // *заголовок программы*  {  *Описание локальных переменных и констант Операторы*  } | Головная функция |
| *Описание функций* |  |

В основу любого компилятора для языка С и С++ заложена однопроходовая схема, т. е. процесс построения программы осуществляется за один проход по тексту. Таким образом, все имена переменных и констант программы должны быть объявлены до их первого использования.

### Препроцессор

Для создания дополнительного сервиса, а также улучшения переносимости программ, в С++ реализован так называемый *препроцессор*, который выпол- няет набор инструкций перед началом компиляции программы. Все директи- вы препроцессора начинаются с символа #.

#### #include

Директива включения файла. Обычно используется для вызова так называе- мых файлов включений, содержащих *прототипы* (т. е. описания) библиотеч- ных функций. Также позволяет включить часть текста программы, записан- ного в другой файл. По сути дела, как в текстовом редакторе, объединяются два файла.

Формат директивы:

#include <имя файла включения>

Здесь угловые скобки < > являются частью конструкции и указывают на то, что файл ищется в каталоге файлов включения, прописанном в среде про- граммирования.

Формат директивы:

#include "имя файла включения"

Двойные кавычки " " означают, что поиск файла включения осуществляется в том же каталоге, в котором открыт исходный файл.

#include <stdlib.h> // Включение стандартной библиотеки #include "meny.h" // Включение файла meny.h

#### #define

Определение символического обозначения. Устаревшая конструкция, вместо нее более целесообразно определять константные переменные, но об этом чуть позже.

Формат директивы:

#define ИМЯ ЗНАЧЕНИЕ

В результате действия директивы все символические обозначения ИМЯ в тек- сте программы заменятся на ЗНАЧЕНИЕ перед началом компиляции. Исключе-

ние составляют текстовые константы, заключенные в двойные кавычки "...", внутри которых подстановки не производятся. Опять же, проводя аналогию с текстовым редактором, это просто операция *"поиск и замена"*, например:

#define PI 3.14159

Перед компиляцией все переменные PI будут заменены их значением

3.14159.



Компилятор языка С++ различает строчные и прописные буквы, поэтому для выделения в тексте программы символических констант их принято писать за- главными буквами.

Некоторые другие директивы мы рассмотрим по мере надобности, но на дан- ном этапе главная директива препроцессора, с которой будут начинаться все наши программы, — это директива включения #include.

### Создание простой программы

С-программа имеет ярко выраженный модульный характер. Ее головная функция состоит из множества обращений к функциям, выполняющим те или иные действия, и отличается от прочих только тем, что имеет фиксированное имя main.

**Листинг 1.1. Простейшая С-программа**

#include <iostream.h> // Включение библиотеки main()

{ int i; /\* Описание целой переменной\*/ for (i = 0; i < 10; i++) // Оператор цикла

cout << i << endl; // Вывод числа

}

Здесь после двойного слэша // помещен комментарий, который занимает ос- таток строки и игнорируется компилятором. Можно также располагать ком- ментарий между парами ограничителей /\*...\*/, в этом случае он может и не ограничиваться одной строкой.

Не вдаваясь пока в детали, прокомментируем приведенный текст программы:

1. Первая строка — подключение файла iostream.h, расположенного в стан- дартном каталоге include среды разработки (на это указывают угловые скобки < >), который должен быть в ней прописан, что обычно происхо- дит при инсталляции компилятора. Этот файл необходим для того, чтобы

можно было воспользоваться оператором потокового вывода cout <<. Во- обще-то cout — это экземпляр класса ostream, но пока не будем углуб- ляться, отметим только, что это работает.

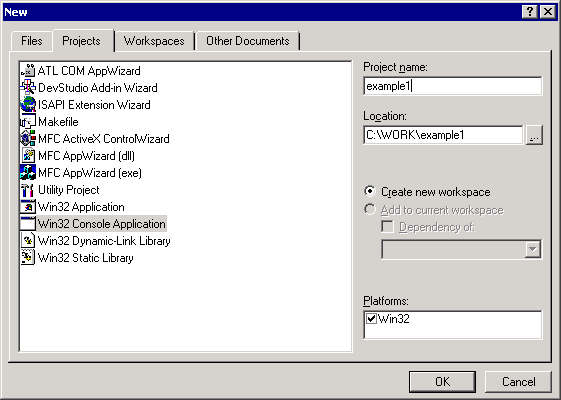
1. main() — имя головной функции. Именно ей, согласно стандарту языка, передается управление при запуске программы. В С-программе должна обязательно присутствовать функция main().
2. Тело функции располагается между открывающей и закрывающей скоб- ками { }. Надо иметь в виду, что компилятор очень скрупулезно подсчи- тывает, сколько скобок открыто и сколько закрыто — если скобку забыли закрыть, он выдает сообщение об ошибке.
3. int i; — описываем переменную для хранения целого числа.
4. for (i = 0; i < 10; i++) — оператор цикла, обеспечивает выполнение оператора вывода на консоль 10 раз, изменяя значение переменной i от 0 до 9.
5. cout << i << endl; — вывод числа на экран монитора (консоль вывода), после чего производится переход в начало следующей строки endl *(end of line)*. Читаем cout как *console output*. Как видно из примера, оператор cout << можно применять агрегатно и выводить сразу несколько значе- ний, а тип выводимого значения оператор определит сам.

Для выполнения рассмотренного примера откроем среду программирования MS Visual C++ 6.0, запустив головной модуль msdev.exe. Пока будем рабо- тать с консольными приложениями.

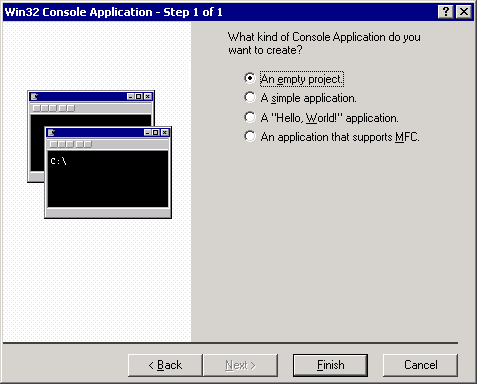
1. Выполнив команду меню **File | New**, выберем **Win32 Console Application**

в окне создания проекта New, открытом на вкладке **Projects** (рис. 1.1).

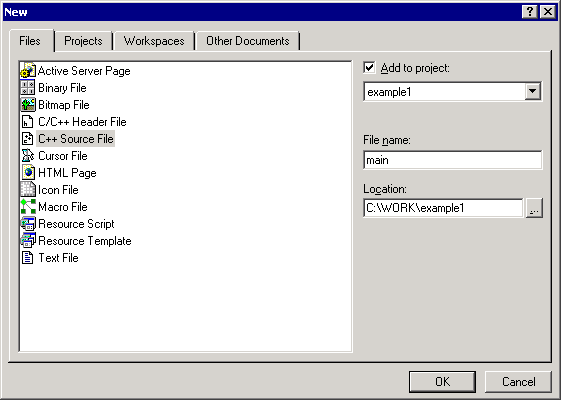
1. В поле **Location** введем вручную (либо воспользуемся поиском по кноп- ке **...**) имя рабочей папки. Например, C:\WORK.
2. В поле **Project name** введем имя нашего будущего проекта — example1, после чего нажмем кнопку **ОК**.
3. Появляется диалоговое окно, приведенное на рис. 1.2, в котором мы оста- вим настройку по умолчанию — **An empty project** (Пустой проект).
4. Получаем диалоговое окно **New Project Information**. Писать программу пока негде, на данном этапе лишь создана папка для нового проекта C:\WORK\example1.
5. Выполнив команду меню **File | New**, выберем на вкладке **Files** пункт **C++ Source File** (рис. 1.3) и в поле **File name** введем имя файла, например main. (Проверьте, что перед полем **Add to project** выставлен флаг, иначе потом придется "вручную" добавлять файл к проекту.)



**Рис. 1.1.** Окно создания проекта

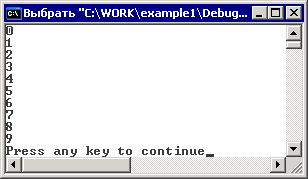


**Рис. 1.2.** Создание проекта Win32 Console Application. Шаг 1



**Рис. 1.3.** Добавление файла к проекту

1. Введем текст программы и запустим на выполнение нажатием кнопки **!**. Это единственная кнопка со значком красного цвета. Получим результат, приведенный на рис. 1.4.



**Рис. 1.4.** Результат выполнения программы

Набирая текст этой программы, сложно допустить ошибку, но если это все- таки произошло, перейдите в окно вывода **Output** на вкладку **Build** и устано- вите указатель на сообщение об ошибке. После двойного щелчка на нем ком- пилятор отметит строку с ошибкой. Исправьте ошибку и повторите процеду- ру запуска программы. Если у вас есть навыки работы в текстовых редакто- рах, то ввод и редактирование текста программы не должны вызывать никаких проблем. Однако, поскольку это все-таки специализированный ре-

дактор, он имеет некоторые дополнительные возможности. Например, если в тексте программы установить курсор рядом со скобкой либо выделить ее, можно использовать одну из следующих полезных команд: поиск парной скобки и выделение блока текста, заключенного в скобки.

Полный список команд редактора можно посмотреть, выполнив команду ме- ню **Help | Keyboard Map**.

## Типы данных. Описание переменных

Вначале рассмотрим подробнее, как хранятся переменные различных типов в памяти. Адресуемая единица памяти, *байт*, состоит из 8 разрядов *(бит)*, куда и записывается число в двоичном коде, например:

00010101 = 24 + 22 + 20 = 16 + 4 + 1 = 21.



*Двоичная система счисления* — позиционная система с основанием 2.

Правила двоичной арифметики очень просты (табл. 1.1).

***Таблица 1.1.*** *Двоичная арифметика*

|  |  |
| --- | --- |
| 0 + 0 = 0 | 0 \* 0 = 0 |
| 0 + 1 = 1 | 0 \* 1 = 0 |
| 1 + 1 = 10 | 1 \* 1 = 1 |

В один байт может быть записан код от 0 до 255 (28–1).

***Таблица 1.2.*** *Перевод двоичных чисел*

*в восьми-, десяти- и шестнадцатеричное представление*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **2** | **16** | **8** | **10** | **2** | **16** | **8** | **10** |
| 0000 | 0 | 0 | 0 | 1000 | 8 | 10 | 8 |
| 0001 | 1 | 1 | 1 | 1001 | 9 | 11 | 9 |
| 0010 | 2 | 2 | 2 | 1010 | A | 12 | 10 |
| 0011 | 3 | 3 | 3 | 1011 | B | 13 | 11 |
| 0100 | 4 | 4 | 4 | 1100 | C | 14 | 12 |
| 0101 | 5 | 5 | 5 | 1101 | D | 15 | 13 |
| 0110 | 6 | 6 | 6 | 1110 | E | 16 | 14 |
| 0111 | 7 | 7 | 7 | 1111 | F | 17 | 15 |

Пользуясь табл. 1.2, можно очень просто представить двоичное число в виде его шестнадцатеричного аналога. На самом деле, числа записывают в двоич- ной форме очень редко, поскольку тогда они занимают много разрядов. Обычно двоичное число записывают более компактно, используя шестнадца- теричную форму записи, пример которой приведен на рис. 1.5.



**Рис. 1.5.** Преобразование двоичного числа в шестнадцатеричное представление

Нетрудно было разбить это число на тетрады и записать, сверившись с таб- лицей кодов, результат. Обратное преобразование также не вызывает слож- ности, достаточно вместо каждой шестнадцатеричной цифры записать соот- ветствующую двоичную тетраду.

Размер выделенной памяти под переменные различных типов и диапазон их изменения приведен в табл. 1.3—1.4.

***Таблица 1.3.*** *Размер типа данных Visual C++*

|  |  |  |
| --- | --- | --- |
| **Тип** | **Размер в битах** | **Диапазон** |
| bool | 8 | true, false |
| char | 8 | –128 .. 127 |
| short | 16 | –32768 .. 32767 |
| int, long | 32 | –2147483648 .. 2147483647 |
| float | 32 | 3.4E-38 .. 3.4E+38 |
| double | 64 | 1.7E-308 .. 1.7E+308 |
| long double | 80 | 3.4E-4932 .. 1.1E+4932 |

Тип bool отсутствовал в классическом С и появился лишь в С++.



 Для 32-битных приложений тип int совпадает с long (полная запись long int).

 Целые типы данных могут иметь модификатор unsigned (беззнаковый).

Диапазон таких переменных отсчитывается от 0.

***Таблица 1.4.*** *Размер целых беззнаковых типов*

|  |  |  |
| --- | --- | --- |
| **Тип** | **Размер в битах** | **Диапазон** |
| unsigned char | 8 | 0 .. 255 |
| unsigned short | 16 | 0 .. 65535 |
| unsigned int, long | 32 | 0 .. 4294967295 |

Рассмотрим подробнее, как в памяти представлена переменная типа short. Например, константа 1 в двоичном представлении имеет вид:

0000 0000 0000 0001.

Чтобы получить число 2, нужно прибавить к 1 еще 1 и т. д., в результате по- лучаются все целые числа.

+ 0000 0000 0000 0001

0000 0000 0000 0001

= 0000 0000 0000 0010

Отрицательные числа для процессора 80 86 представлены в дополнительном коде и образуются инверсией числа с добавлением 1 (при инвертировании кода 1 заменяется на 0, а 0 на 1), например –1 получается по следующей схеме:

|  |  |  |  |
| --- | --- | --- | --- |
| + 1111 | 1111 | 1111 | 1110 + ~1 |
| 0000 | 0000 | 0000 | 0001 1 |

= 1111 1111 1111 1111 = -1

Такой способ образования отрицательных чисел позволяет свести операцию вычитания к операции сложения с отрицательным числом. Таким образом, операция вычитания является излишней для системы команд процессора и может быть исключена. Пример: 1 – 1 = 1 + (–1) = 0.

+ 0000 0000 0000 0001

1111 1111 1111 1111

= 0000 0000 0000 0000

С++ относится к таким языкам программирования, в которых все перемен- ные, используемые программой, должны быть предварительно описаны. В описании переменной задается ее класс памяти и тип для того, чтобы транслятор выделил нужное количество памяти и в нужном разделе. Для обо- значения переменной служит имя — символическое обозначение перемен- ной. Начинается имя с буквы или символа подчеркивания "\_", может содер-

жать буквы латинского алфавита, цифры и знак подчеркивания. Длина имени может быть произвольной, но распознается по первым 32 символам (это за- висит от настроек компилятора). Строчные и прописные буквы различаются компилятором как разные буквы.

Переменные могут быть описаны в любом месте текста программы, но обяза- тельно до их использования. Имена переменных в операторах описания отде- ляются запятыми. Пример:

int i,j,k;

Возможна инициализация переменных при описании, например:

int i = 0;

**Переменные целого типа: *bool*, *char*, *short*, *int*, *long***

При описании переменной целого типа необходимо учитывать диапазон ее изменения. Если переменная может принимать одно из двух значений (true, false), используется тип bool. Если же необходим больший диапазон выбора значений, можно использовать переменную одного из следующих типов: char (от –127 до +127), short (от –32 768 до +32 767) или int (от –2 147 483 648 до

+2 147 483 647). Тип long совпадает с типом int. Однако для переменных це- лого типа необходимо иметь в виду, что при выполнении арифметических операций процессор разворачивает число до длины машинного слова (для 32-битных приложений удвоенное машинное слово — 4 байта), т. е. никакой экономии в скорости выполнения операций не достигается при использова- нии типа меньшей размерности, так что выбор типа данных определяется лишь экономией памяти при хранении данных.

Если заведомо известно, что переменная принимает только положительные значения, целесообразно использовать модификатор unsigned, который обес- печит дополнительный контроль со стороны компилятора на присваива- ние отрицательного значения и увеличит вдвое диапазон положительных чисел.

***enum* — перечисляемый тип**

*Перечисляемый тип* используется, как правило, для описания символических констант. Переменным, перечисленным в списке enum, присваиваются после- довательно целые значения, начиная с 0. Пример:

enum { BLACK, BLUE };

Здесь описан неименованный объект перечисляемого типа, в котором опре- делены две символические константы: BLACK = 0 и BLUE = 1.

Можно, однако, описать переменную и присваивать значение в списке enum

явно, например:

enum io\_state { goodbit = 0x00,

eofbit = 0x01, failbit = 0x02, badbit = 0x04 };

В данном контексте переменная io\_state может принимать только значения, указанные в перечислении.

### Переменные с плавающей точкой:

***float*, *double*, *long double***

Внутреннее представление переменных с плавающей точкой более сложно. Число представляется в виде: 0.m\*2p, где первый сомножитель *(мантисса)* лежит в диапазоне от 0.5 до 1, а порядок p, чтобы не вводить знаковый раз- ряд, записывается с дополнением.

#### Основные типы переменных с плавающей точкой

1. float занимает 4-байтную область памяти, которая делится на три поля:

Знаковый разряд — 1 бит.



Порядок с дополнением +127 — 8 бит. Мантисса — 23 бита.

|  |  |  |
| --- | --- | --- |
| 1 | 8 | 23 |
| знак | порядок+127 | мантисса |

Так, например, число 0,75\*22 может быть представлено в виде: знак + = 0, порядок *p* = 127+2=129, мантисса *m* = 0.75= ½ + ¼.

На самом деле ситуация еще интересней. Если мантисса больше 0.5, то первый разряд мантиссы будет всегда заполнен, а если это так, то нет ни- какого смысла хранить этот разряд в памяти. Так и происходит, первый разряд мантиссы при записи числа исключается, а перед операцией про- цессор автоматически добавляет этот разряд (½ не отображается).

|  |  |  |
| --- | --- | --- |
| 0 | 10000001 | 100000000000000000000000 |



С этим связано разночтение у некоторых авторов. Иногда пишут, что мантисса лежит в диапазоне от 0 до 0.5.

1. double занимает 8-байтную область, которая также делится на три поля. Знаковый разряд — 1 бит.



Порядок с дополнением +1023 — 11 бит. Мантисса — 52 бита.

|  |  |  |
| --- | --- | --- |
| 1 | 11 | 52 |
| знак | порядок+1023 | мантисса |

Особенность типа double заключается лишь в увеличении диапазона для порядка до 11 разрядов и 52 разряда под мантиссу, что позволяет увели- чить точность записи числа. Так, если для типа float числа записываются с точностью до 5—6 значащих цифр, то для double — с точностью до 15—16 значащих цифр.

1. long double имеет расширенный диапазон 3.4е-4932 .. 1.1е+4932 и зани- мает 10-байтную область. Этот тип используется редко.

Своеобразие арифметики с плавающей точкой в языке С++ заключается в том, что для повышения точности вычислений компилятор для перемен- ных типа float автоматически ставит преобразование в double перед опе- рацией, а результат приводит вновь к типу float. Таким образом, исполь- зование типа float оправдано лишь экономией памяти при хранении дан- ных, но приводит к "буйному" преобразованию типов, что вряд ли следует приветствовать. Поэтому мы в большинстве случаев будем использовать тип double для данных с плавающей точкой.

### Целые константы

*Целые константы* могут быть записаны в десятичном, восьмеричном и ше- стнадцатеричном представлении. Отличительным признаком восьмеричной константы является первый символ "0", а шестнадцатеричной — "0х".

Пример:

* 2, 5, –18 — десятичные константы;
* 076, –0237 — восьмеричные константы;
* 0xf1, 0x4ea3 — шестнадцатеричные константы.

Знак "–" перед константой рассматривается как унарная операция "минус".



Унарная операция имеет 1 операнд, а бинарная, соответственно, два.

По умолчанию целые константы конструируются компилятором типа int. Суффикс u (U) определяет модификацию типа unsigned, а l (L) — long.

Пример:

129u, 0xf0e1u; 2l, 0xffl;

### Константы с плавающей точкой

Если константа содержит десятичную точку "." или символ экспоненты "e", то она рассматривается как *константа с плавающей точкой* типа double. Константа типа float может быть получена добавлением суффикса f.

Пример:

1.2e-2, .85, 10., 1e-6f;

Константа 1.2e-2 интерпретируется как 1.2\*10-2.

### Символьные константы

*Символьные константы* состоят из символа, заключенного в одинарные ка- вычки, например: 'a', 'b', '0'. Константа является целым числом, равным коду символа. Допускается использование четырехсимвольных констант, та- ких как 'abcd', которые занимают удвоенное машинное слово (4 байта), при этом первый символ находится в младшем байте, а второй — в старшем и т. д.



Зависит от размера типа int, в Turbo C допускаются 2-символьные константы.

Для ввода специальных символов в С++ зарезервированы управляющие сим- вольные константы, перечисленные в табл. 1.5.

***Таблица 1.5.*** *Управляющие символьные константы*

|  |  |
| --- | --- |
| \a | Звуковой сигнал |
| \b | Возврат курсора на одну позицию |
| \f | Перевод страницы |
| \n | Перевод строки |

***Таблица 1.5*** *(окончание)*

|  |  |
| --- | --- |
| \a | Звуковой сигнал |
| \r | Возврат каретки |
| \t | Табуляция |
| \v | Вертикальная табуляция |
| \\ | Обратный слэш |
| \' | Одинарная кавычка |
| \" | Двойная кавычка |
| \? | Знак вопроса |
| \bbb | Любой символ, где bbb — восьмеричное число |
| \xhh | Любой символ, где hh — шестнадцатеричное число |

### Строковые константы

Любая последовательность символов, заключенная в двойные кавычки, на- пример "текст", рассматривается как *текстовая (строковая) константа*. В С++ принято считать признаком конца текста символ '\0', который добав- ляется автоматически. Таким образом, любая текстовая константа занимает в памяти на 1 байт больше количества символов. В C++ также предусмотрена неявная конкатенация, т. е. "склеивание" строк.

Так, для константы:

"строка 1" "строка 2";

компилятор построит следующую строку:

"строка 1строка 2".

Для переноса длинной строковой константы в тексте программы необходимо завершить переносимую строку символом "\".

Пример 1.

"Очень длинный текст, который хотелось бы \ разместить в одной строке"

Пример 2.

#include <iostream.h> main()

{

int i, j = 6, k = -1; char h = '0', ch;

double pi = 3.14; i = j + k;

cout << "int i=j+k: " << i << endl; ch = h + 1;

cout << h << '\t' << ch << endl; cout << "pi: " << pi << endl;

}

Первая строка, как обычно, подключает стандартную библиотеку потокового ввода/вывода. Далее в функции main() описываем переменные целого типа:

int i, j = 6, k = -1;

Причем первая переменная объявлена, но значение ей не присвоено (значе- нием может быть любой "мусор"), второй переменной присвоено значение 6, а третьей –1.

char h = '0', ch; // Две переменных типа char. double pi = 3.14; // Одна переменная double,

// инициализированная значением 3.14

i = j + k; // Обычная операция сложения, об этом чуть позже. cout << "int i=j+k: " << i << endl;

Здесь выводим текстовую строку и переменную i. Отметим также, что опе- ратор вывода << корректно выведет данные различного типа. Пока воспри- нимаем это как данность, с потоковым выводом мы будем разбираться позже.

ch = h + 1;

А вот здесь интересная операция, к символу '0' прибавили 1. Ничего уди- вительного, наверное, нет в том, что в результате получится символ '1', но, пожалуй, ни один другой язык программирования не позволит такой опера- ции.

cout << h << '\t' << ch << endl;

Чтобы убедиться в этом, посмотрите результат вывода двух символов. Мы вставили между ними разделитель "горизонтальная табуляция" и завершили вывод специальным манипулятором endl (end of line), который обеспечит перевод строки. Он описан в файле включений iostream.h.

cout << "pi: " << pi << endl;

И наконец, вывод числа с плавающей точкой.

### Переменные-константы

Когда мы определяем в программе константу, мы, как правило, хотели бы быть уверены, что значение этой константы случайно не изменилось. Конеч-

но, по большей части такие проблемы остаются на совести программиста, но компилятор дает возможность обеспечить дополнительный контроль над из- менением значений переменных. Для этого служит модификатор const, кото- рый ставится перед указанием типа переменной. Например, можно было бы написать:

const double pi = 3.14; const int NULL = 0;

Теперь, при попытке изменить значение константной переменной, будет вы- дано сообщение об ошибке.

NULL = 1; // Ошибка

### Комментарии

Для включения комментария в текст программы на языке С++ предусмотре- ны две конструкции:

* Если *комментарий* занимает одну строку, то достаточно поставить два символа // (двойной слэш) и текст, расположенный далее до конца строки, будет считаться комментарием и игнорироваться компилятором.
* Если же комментарий занимает несколько программных строк, он ограни- чивается с обеих сторон парой символов /\* . . . \*/.

// Простой комментарий

/\*Комментарий,

занимающий несколько строк \*/

### Классы памяти

Как правило, программиста не заботит проблема, в какой области памяти компилятор разместит переменные, но иногда возникает необходимость управлять этим процессом. По умолчанию компилятор строит переменные *класса памяти* auto — это автоматические переменные, которые создаются в стеке при входе в функцию и освобождаются при выходе из нее, причем на- чальные значения таких переменных могут быть произвольными. Перемен- ные класса static располагаются в фиксированном блоке памяти, выделяе- мом перед началом выполнения программы, инициализируются нулевым значением. Время жизни таких переменных совпадает с временем жизни про- граммы. И наконец, для переменных целого типа, определен весьма специ- фичный класс register. В этом случае компилятор пытается выделить сво- бодный регистр, и если это удается, операции с такой переменной будут осу-

ществляться существенно быстрее, если же свободного регистра нет, строит- ся обычная переменная класса auto.

static const int NULL = 0; register int k;

## Операторы и операции

Основной *операцией* любого языка программирования является *операция присваивания*:

операнд = выражение;

i = j + k;

В отличие от других языков программирования, в С++ допускается много- кратное присваивание, причем операция выполняется справа налево, напри- мер:

i = j = k = 0;

или так:

i = 2 + (k = 3);

Во втором случае переменная i принимает значение 5 и, попутно, переменная k принимает значение 3, т. к. в языке С++ принято, что скобка имеет значение последней выполняемой в ней операции.

### Арифметические операции

В языке имеется набор обычных математических операций, выполнение ко- торых осуществляется в соответствии с общепринятыми правилами соблю- дения приоритетов. Причем операции одного приоритета, например умноже- ние и деление, выполняются слева направо.

* \* — умножение
* / — деление
* % — остаток от деления (для целых)

 + — сложение

* - — вычитание

Операция % корректно работает с положительными значениями, но для от- рицательных чисел ее результат не очевиден и зависит от компилятора.



 Необходимо иметь в виду, что в языке С++ отсутствует операция возведе- ния в степень, однако имеется функция, реализующая данную операцию.

### Приведение типов

При выполнении арифметических операций результат зависит от типов опе- рандов. Так, если операнды целого типа, то и результат будет целого типа. Однако часто возникает ситуация, когда операнды принадлежат разным ти- пам. В этом случае работает правило *приведения типа*, согласно которому типы операндов приводятся к более общему типу.



При совершении арифметической операции с переменными типа char и short, они разворачиваются до базового типа int.

Так, в выражении:

double avg, sum; int n;

. . .

avg = sum/n; double num = n;

переменная типа double делится на переменную типа int. Здесь компилятор вставит *автоматическое преобразование типа* int в double, который являет- ся более общим типом. То же самое произойдет и в следующей операции присваивания.

Однако часто бывает необходимо *явно преобразовать тип данных*, например, если мы хотим получить результат от деления двух целых чисел:

int a = 3, b = 2; double r = a/b;

то мы получим r = 1.0 вместо 1.5. Дело в том, что вначале происходит деле- ние двух целых чисел, результат которого 1, и лишь затем это значение будет приведено к типу double. Для того чтобы деление произошло над данными типа double, необходимо хотя бы один из операндов привести к типу double. Для этого нужно воспользоваться оператором явного преобразования типа одним из трех способов:

* (double)a — базовый оператор классического С.
* double(a) — расширение языка, преобразование типа как функция.
* static\_cast<double>(a) — современный стиль.

Если мы напишем выражение:

double r = double(a)/b;

получим: r = 1.5;

**Операции ++ и --**

Специфичные для языка С++ *операции приращения* для переменной целого типа char, short, int, long приводят к увеличению ++ (increment) или уменьшению –- (decrement) значения переменной на 1. Если данная опера- ция стоит перед переменной, то она будет произведена перед использованием переменной *(префиксная операция)*, если после переменной — после ее ис- пользования *(постфиксная операция)*.

Именно наличием этих операций объясняется название языка С++ (следую- щий шаг после С). Эти операции сводятся к одной машинной команде и вы- полняются с большей эффективностью, чем операции сложения и вычитания.

i = 0;

j = ++i; // j = 1, i = 1

k = i--; // k = 1, i = 0

### Битовые операции

Осуществляются над переменными, рассматриваемыми как битовые цепочки, и применяются для целого типа char, short, int, long. *Битовая операция* производится над каждым разрядом без переносов.

***Таблица 1.6.*** *Битовые операции*

|  |  |  |
| --- | --- | --- |
| & | Битовое умножение  Пример: (10010011) & (00111101) = (00010001) | 1 & 1 = 1  0 & 1 = 0  0 & 0 = 0 |
| | | Битовое сложение (ИЛИ)  Пример: (01101110) | (10000000) = (11101110) | 1 | 1 = 1  0 | 1 = 1  0 | 0 = 0 |
| ^ | Битовое исключающее сложение (исключающее ИЛИ) Пример: (01010101) ^ (11111111) = (10101010) | 1 ^ 1 = 0  0 ^ 1 = 1  0 ^ 0 = 0 |
| ~ | Битовое отрицание (НЕ)  Пример: ~(10011010) = (01100101) | ~1 = 0  ~0 = 1 |

***Таблица 1.6*** *(окончание)*

|  |  |  |
| --- | --- | --- |
| << | Логический сдвиг влево  Все разряды переменной сдвигаются на указанное количество разрядов влево, разряды, выходящие за разрядную сетку, те- ряются, а освобождающиеся разряды справа устанавливаются в 0.  Пример: (00000001) << 4 = (00010000) |  |
| >> | Логический сдвиг вправо  Все разряды переменной сдвигаются на указанное количество разрядов вправо, разряды, выходящие за разрядную сетку, те- ряются, а освобождающиеся разряды слева устанавливаются в 0.  Пример: (00010000) >> 4 = (00000001) |  |

Примеры использования битовых операций:

i << 1; // умножить на 2

i >> 2; // поделить на 4

i & 1; // проверка на нечетность i | 128; // установить в 1 7-й бит i ^ 128; // инвертировать 7-й бит

~i; // инвертировать все биты



Умножение и деление нацело на число, кратное 2, при помощи операций сдви- га корректно только для положительных значений.

### Комбинированные операции

Часто возникает необходимость осуществлять операции с одной переменной в левой и правой части операции присваивания. В этом случае удобно ис- пользовать так называемые *комбинированные операции*. Например, вместо операции i = i + 2 можно написать i += 2, что имеет такой же смысл, но облегчает компилятору построение более эффективного программного кода.

Приведем список комбинированных операций:

|  |  |
| --- | --- |
| i += j; | // i = i + j; |
| i -= j; | // i = i - j; |
| i \*= j; | // i = i \* j; |
| i /= j; | // i = i / j; |
| i %= j; | // i = i % j; |
| i <<=j; | // i = i <<j; |

i >>=j; // i = i >>j;

i &= j; // i = i & j;

i |= j; // i = i | j;

i ^= j; // i = i ^ j;

### Операции отношения

Для сравнения значений переменных используются *операции отношения*:

|  |  |  |
| --- | --- | --- |
| > | больше | (a > b) |
| >= | больше или равно | (a >= b) |
| < | меньше | (i < 0) |
| <= | меньше или равно | (i <= j) |
| == | равно | (i == k) |
| != | не равно | (ch != 'y') |



Равенство выполнено двумя знаками равно. Типичная ошибка начинающего программиста, когда вместо равенства записывается операция присваивания =.

В классическом С не было логических переменных, они появились лишь в С++. Однако и при отсутствии логических переменных язык С обходился без больших проблем. Было принято считать, что результат имеет значение ИСТИНА, если он отличен от 0, и ЛОЖЬ в противном случае. В С++ можно ис- пользовать как логические переменные, так и подход классического С, когда вместо логических выражений применяются арифметические значения. На- пример, для проверки нечетности целого значения можно воспользоваться выражением (i & 1) != 0 или просто (i & 1).

В первом случае получаем логическое выражение, равное true, если пере- менная i равна нечетному числу, во втором случае получаем просто 1, что и так ИСТИНА. Вообще, в языке С++ считается дурным тоном сравнивать с 0.

### Логические операторы

*Логические операторы* применяются над логическими операндами, в качест- ве которых могут выступать операции отношения или те же логические опе- раторы (табл. 1.7).

***Таблица 1.7.*** *Логические операторы*

|  |  |  |
| --- | --- | --- |
| && | И | ( i > j) && ( k != 1) |
| || | ИЛИ | (ch == 'y') || (ch == 'Y') |
| ! | НЕ | !(i > 1) |



Обычно компилятор оптимизирует код операции логического умножения, и если первый операнд имеет значение false, то и результат операции && также будет false, так что второе выражение можно и не вычислять.

С помощью логических операторов можно конструировать сложные логиче- ские выражения. Нужно иметь в виду, что операторы языка выполняются в соответствии с их приоритетами, таблица приоритетов помещена в *приложе- нии*, но если есть сомнения в порядке выполнения операций, можно исполь- зовать скобки.

## Операторы языка С++

Оператор может быть простым либо составным. *Простой оператор* заканчи- вается символом точка с запятой ";" — частью оператора, отличающейся С++ от других языков программирования, где символ ";" — разделитель операторов. *Составной оператор* представляет собой последовательность простых операторов, заключенную в фигурные скобки.

Каждый оператор может иметь метку, которая используется оператором пе- рехода goto. *Метка* — это имя, за которым стоит двоеточие ":". Областью действия метки является текущая функция. В С++ метки предварительно не описываются.

{ int i; home: i = 0;

. . .

}

### Оператор перехода *goto*

*Оператор перехода* служит для безусловной передачи управления оператору с данной меткой в пределах текущей функции:

goto метка;

Хорошо структурированная программа не должна вызывать необходимости использования оператора перехода, поэтому в современном стиле програм- мирования считается дурным тоном использование операторов goto. Однако они могут быть весьма полезны для обработки критических ситуаций или выхода из глубоко вложенных циклов.



Сейчас для обработки критической ситуации используется механизм исклю- чений.

goto end;

...

end: exit(1);

**Условный оператор *if . . . else***

Назначение *условного оператора* заключается в организации ветвлений про- граммы в зависимости от условия. Используются две формы записи данного оператора.

Сокращенная форма записи:

if (логическое выражение) оператор;

Оператор выполняется, если логическое выражение истинно, иначе выполня- ется следующий по тексту программы оператор.

Или полная форма записи:

if (логическое выражение) оператор 1; else оператор 2;

Если выражение истинно, выполняется оператор 1, иначе — оператор 2, на- пример:

if ( i & 1) a = 0; else a = 1;

// Если i нечетно, то переменной a присваивается 0, иначе 1

### Условный арифметический оператор

Последний пример выглядит несколько коряво. В самом деле, для выполне- ния простого присваивания пришлось написать оператор два раза. К счастью, в С++ есть механизм для более элегантного решения этой задачи, так назы- ваемый *условный арифметический оператор*. Его формат:

(условие)? выражение 1 : выражение 2;

если условие истинно, то результатом оператора будет выражение 1, иначе —

выражение 2. Так предыдущий пример можно записать:

a = (i & 1)? 0 : 1;

Числовая константа является частным случаем арифметического выражения.

### Операторы цикла

Для повышения гибкости языка в С++ используются три *оператора цикла*.

#### Цикл с предусловием *while*

while (логическое выражение) оператор;

Оператор цикла выполняется до тех пор, пока логическое выражение истин- но. Если условие сразу не выполняется, оператор ни разу не выполнится. На- пример:

a = i = 0;

while(i < n) a += ++i;

**Цикл с постусловием *do-while***

do оператор;

while (логическое выражение);

Оператор цикла выполняется до тех пор, пока логическое выражение истин- но. Оператор будет выполнен хотя бы один раз.

s = i = 0; do

s += ++i;

while(i < n);

**Цикл *for***

for(Нач.установки; Лог.выражение; Приращение переменных) оператор;

* Начальные установки — позволяют производить начальную инициализа- цию переменных.
* Логическое выражение — контролируется в начале каждой итерации, в слу- чае его истинности тело цикла выполняется. Если логическое выражение ложно при входе в цикл, тело цикла не выполнится ни разу.
* Приращение переменных — происходит после завершения тела цикла.

Если некоторые из выражений в скобках отсутствуют, то символ ";" все рав- но должен присутствовать. Пример:

for (i = 0, s = 0.0; i < n; i++) s += x[i];

В операторе цикла могут отсутствовать все три выражения for(;;) {...}, в этом случае выход должен быть организован внутри тела цикла.



 При выходе из цикла переменная цикла сохраняет свое последнее зна- чение.

 Здесь использована специфичная для языка С++ операция *запятая*. Смысл этой операции заключается в том, что выражения, перечисленные через за- пятую, представляют собой один оператор, а именно один оператор должен стоять в блоке начальной инициализации.

### Простая программа с циклами

В качестве примера составим программу вычисления чисел Фибоначчи (лис- тинг 1.2). Числа образуются согласно следующей рекуррентной формуле:

*fn*



*fn* 1

*fn* 2 ; *f*0 1, *f*1 1.

Каждое число есть сумма двух предыдущих, а два первых значения равны 1.

**Листинг 1.2. Задача нахождения чисел Фибоначчи**

#include <iostream.h> void main()

{ int f0, f1, f, i, n;

cout << "Fibonacci series? "; cin >> n;

for (f = f1 = 1, i = 2; i <= n; i++)

{

f0 = f1; f1 = f;

f = f0 + f1;

cout << i << '\t' << f << endl;

}

}

Результат выполнения программы:

Fibonacci series? 10

|  |  |
| --- | --- |
| 2 | 2 |
| 3 | 3 |
| 4 | 5 |
| 5 | 8 |
| 6 | 13 |
| 7 | 21 |
| 8 | 34 |
| 9 | 55 |
| 10 | 89 |

Здесь мы используем оператор консольного ввода cin, который выглядит аналогично оператору вывода, но символы >> показывают направление пере- дачи данных к переменной cin >> n;.

**Вспомогательные операторы: *break*, *continue***

Эти операторы предназначены для повышения гибкости языка и используют- ся в операторах цикла, а оператор break и в переключателе switch. Так опера- тор break приведет к выходу из оператора цикла или переключателя, а опера- тор continue — к переходу для выполнения следующей итерации.

for (i = k = 0; i < n; i++)

{

if ( x[i] < 0) continue;

if ( x[i] == 0) break; // Выход, если нуль

k++; // Подсчет количества положительных значений

// до первого нуля

}

**Переключатель *switch***

Переключатель switch является удобным способом организации альтерна- тивных процессов в зависимости от значения ключевого арифметического выражения.

switch (арифметическое выражение) { case константа 1 : операторы 1;

...

case константа n : операторы n; default : операторы;

}

Здесь происходит сравнение выражения и констант. Начинают выполняться операторы той строки, константа которой совпадает с выражением. Если же ни одна из констант не совпала с арифметическим выражением, выполняются операторы строки альтернативы default, если же отсутствует строка default, то не выполняется ни одного оператора в переключателе и управление пере- дается следующему за переключателем оператору.



В отличие от аналогичной конструкции языка Паскаль, здесь не происходит ав- томатического выхода из переключателя после выполнения всех операторов строки case, поэтому каждая строка обычно заканчивается оператором break. Иначе будут выполняться следующие строки программного кода в переключа- теле.

Пример:

switch (ch) {

case '+': c = a + b; break;

case '-': c = a - b; break; default : c = 0;

}

В качестве примера сделаем простой калькулятор, выполняющий четыре арифметические операции (листинг 1.3).

|  |
| --- |
| **Листинг 1.3. Программа "Калькулятор"** |

#include <iostream.h> void main()

{ double a, b, c; char ch;

cout << "Calculator" << endl;

cout << "Example: a + b <Enter>\n";

cin >> a >> ch >> b; // Оператор ввода также применяется агрегатно switch (ch)

{ case '+': c = a + b; break; case '-': c = a - b; break; case '\*': c = a \* b; break; case '/': c = a / b; break;

default: cout << "Error operator " << ch << endl;

}

cout << c << endl;

}

## Массивы и указатели

Для решения многих задач требуется вводить набор однотипных данных и, естественно, в языке программирования предусмотрена такая конструкция, как *массив*. Описывается массив следующим образом:

тип имя[размер];

В качестве типа элементов массива может использоваться любой стандарт- ный тип, а также и пользовательский тип. Имя массива образуется по тем же правилам, что и имя простой переменной, а размер — целое число или пере- менная, значение которой компилятор может определить (константная пере- менная). Дело в том, что под массив компилятор должен выделить память, и размер этой памяти ему должен быть известен.

const int N = 100;

int a[10]; // Описан массив из 10 чисел типа int double x[N]; // Описан массив из 100 чисел типа double

Доступ к элементам массива осуществляется при помощи индекса:

a[0], a[1], ... a[9]

Нужно учитывать, что индексы в С++ отсчитываются от 0, поэтому если раз- мер массива N, то последний *индекс* будет N–1. Если при описании массива используется переменная класса памяти auto, то при загрузке программы со- держимое элементов массива не определено, поэтому, прежде чем использо- вать элементы массива в вычислениях, им нужно присвоить некоторые зна- чения. При описании же массива в статической области памяти static на- чальные значения элементов массива будут нулевыми. Можно также присвоить значения элементам массива прямо при описании:

тип имя[размер] = {первый элемент, второй элемент,...};

Пример:

int a[10] = {1, 2, 3, 4, 5};

Здесь описан массив из 10 чисел типа int, где первым 5 числам будет при- своено значение. Возникает вопрос — а чему равны остальные 5 элементов массива? Оказывается, компилятор присвоит им нулевые значения. Можно при присваивании значений элементам массива вообще не указывать размер массива, в этом случае компилятор сам подсчитает количество инициализи- рующих значений и создаст массив необходимого размера, например:

int a[] = {1, 2, 3, 4, 5};

Здесь будет создан массив из 5 значений типа int. Или такое описание:

static double arr[16]; // Массив инициализирован 0.0

### Примеры программ с использованием массивов

Рассмотрим несколько типичных программ с использованием массивов (лис- тинги 1.4—1.6):

|  |
| --- |
| **2**  **Листинг 1.4. Табулирование функции y(x) = e-x на отрезке [-1;1]**  **с шагом h=0.2** |

#include <iostream.h>

#include <math.h> // Здесь описана функция exp() void main()

{ const int N = 100; int i;

double x, h, a = -1.0, b = 1.0, y[N+1]; h = (b - a)/N;

for (i = 0, x = a; i <= N; i++, x += h)

{

y[i] = exp(-x\*x);

cout << x << '\t' << y[i] << endl;

}

}

**Листинг 1.5. Суммирование элементов массива**

#include <iostream.h> void main()

{

int i, s, k[] = {2, 4, 5, 7, 9, 11, 12, 14};

for (i = s = 0; i < 8; i++) s += k[i]; cout << "sum: " << s << endl;

}

Здесь нам пришлось подсчитать размер массива, чтобы написать условие за- вершения цикла i < 8. Однако можно возложить эту работу на компилятор и воспользоваться функцией sizeof(k), которая вернет размер массива в бай- тах. Если этот размер поделить на размер одного элемента, мы получим ко- личество элементов массива. Условие сейчас записывается:

i < sizeof(k)/sizeof(int)

Такую конструкцию можно использовать без риска понижения производи- тельности вычислений, поскольку функция вычисляется на этапе компи- ляции.



Можно использовать и операторную запись sizeof k.

**Листинг 1.6. Вычисление минимального и максимального значения в массиве данных**

#include <iostream.h> void main()

{ double min, max, m[5] = { 1.2, -3, 2.8, 8, 4.0 }; int i;

cout << "Minimum & Maximum" << endl;

for (i = 1, min = max = m[0]; i < 5; i++)

{

if (m[i] > max) max = m[i]; else if (m[i] < min) min = m[i];

}

cout << "min: " << min << endl; cout << "max: " << max << endl;

}

Здесь мы просто организовали перебор всех значений массива и сравнение с переменными min, max, которые предназначены для хранения минимального и максимального значения. Важно, что мы им присвоили в качестве начального значения первый элемент массива (в принципе, можно выбрать любые значе- ния, принадлежащие массиву). Если этого не сделать, мы не сможем пра- вильно решить задачу, поскольку начальные значения могут изначально ока- заться либо больше, либо меньше всех значений, принадлежащих массиву. А поскольку нет смысла сравнивать значение само с собой, цикл мы начина- ем с 1.

### Указатели

Что же такое указатели и почему они являются кошмаром для программи- стов? Оказывается, *указатели* — это просто адреса памяти, указывающие на соответствующие переменные. Описываются указатели при помощи сим- вола \*:

тип \*имя;

Указатели описываются обязательно с указанием типа. Это необходимо для поддержки арифметики указателей, например:

int \*q;

Здесь описан указатель на переменную целого типа, причем компилятор вы- деляет память только под указатель (адрес), для хранения самой переменной памяти не выделено. Теперь можно установить указатель на существующую переменную при помощи *операции взятия адреса (*&*)*.

int \*q, r = 1;

q = &r; // q – адрес переменной r, где хранится число 1

Для того чтобы получить значение, на которое указывает указатель, исполь- зуется *операция разадресации* с помощью все того же символа '\*'. Получается какая-то тавтология, но такова терминология в языке С++. Сейчас можно, например, совершить операцию:

int k = \*q;

Здесь переменная k будет равна 1, поскольку ей присваивается значение пе- ременной r, на которую указывает указатель q. Таким образом, r и \*q являют- ся одной и той же переменной.

Посмотрим сейчас, каким образом указатели связаны с массивами, и что это нам дает. Так, имя массива — это просто *константный указатель*, который указывает на первый байт начала массива. Напомним, что константная пере- менная не может изменить своего значения. Таким образом, мы можем при- своить указателю имя массива и оперировать с массивом посредством указа- теля, но не можем присвоить имени массива другое значение. Например, опишем массив и указатель:

int a[5],\*q = a; // a и q — указатели первого элемента массива a[0]

От указателей было бы мало толку, если бы не арифметика указателей. Мож- но над указателями совершать следующие операции (пример программы, де- монстрирующей их выполнение, приведен в листинге 1.7):

* *Сравнение указателей*. Например, если имеется два указателя p и q, то можно проверить их на равенство p == q или на неравенство p!= q. Разу- меется, типы указателей должны совпадать. Очень часто сравнивают ука- затель с нулевым значением: p!= 0.
* *Сложение указателя с числом*. Что означает выражение q+1? Оказывается это указатель следующего элемента массива. То есть увеличение указателя на единицу означает увеличение адреса переменной не на один байт, а на размер соответствующего типа данных. Так, если у нас тип int занимает 4 байта, то и значение указателя увеличится на 4. Таким образом, можно считать эквивалентными следующие выражения:

a[0] \*q;



a[1] \*(q+1);

. . .

a[n-1] \*(q+n-1);

* *Вычитание из указателя* целого числа. Аналогично предыдущему пункту, но указатель здесь сдвигается не вперед, а назад на заданное количество данных соответствующего типа.
* *Операции ++ и --*. Причем поддерживаются как префиксные, так и пост- фиксные операции: ++q, q++. То есть приращение указателя может осуще- ствляться как до его использования, так и после.
* *Разность двух указателей*. Эта операция дает нам количество элементов соответствующего типа в промежутке между двумя указателями. Разуме- ется, указатели должны принадлежать одному массиву данных, иначе опе- рация теряет смысл.

**Листинг 1.7. Пример программы, демонстрирующей операции над указателями**

#include <iostream.h> void main()

{ char \*t = "text"; char \*q = t;

cout << "Pointer \*q" << endl;

while(\*q) cout << \*q++; // Вывод строки посимвольно cout << "\nLength: " << q - t << endl;

}

Эта программа требует некоторых комментариев. Так, в строке:

char \*t = "text";

мы описываем указатель типа char и присваиваем его текстовой строке. Компилятор выделяет область памяти под текстовую константу и присваива- ет ее адрес указателю t. Здесь указатель t не является именем массива, и мы могли бы не вводить дополнительного указателя, но тогда мы потеряем адрес текстовой константы и не сможем больше ею воспользоваться, что не являет- ся хорошим решением. Поэтому мы ввели еще один указатель и установили его на ту же текстовую константу:

char \*q = t;

после чего можем использовать его для продвижения по строке.

Отметим также, что в выражении \*q++ первой выполнится операция разадре- сации, после чего указатель перемещается на следующий байт (см. в *прило- жении* таблицу приоритетов операций).

В операторе while мы довольно странно записали условие (\*q), но если вспомнить соглашение языка С++, по которому любое ненулевое значение считается истиной, то все становится ясно. Действительно, \*q принимает по- следовательно значения 't', 'e', 'x', 't', '\0', и как только \*q == '\0', цикл завершится, поскольку нулевое значение интерпретируется как ложь. В языке С++ именно такой стиль программирования считается общепри- нятым.

И наконец, в записи комментария "\nLength: " мы использовали предопреде- ленную константу '\n' (перевод строки), чтобы последующий вывод пошел с новой строки.

Ответим, в заключение, на следующий вопрос: почему указатели очень опас- ны в использовании? Дело в том, что для повышения эффективности С-программ во время выполнения не проводится контроль выхода указателя за границу массива, поэтому типичная ошибка заключается в выходе указате-

ля за допустимый диапазон. А с помощью указателя можно попасть как в системную область памяти, так и в область другого работающего приложения и испортить там данные.



Windows NT обеспечивает дополнительный контроль и не позволит перейти в адресное пространство другого приложения.

### Определение псевдонимов

Для того чтобы текст программы выглядел более читаемым, часто бывает удобно вводить *псевдонимы имен*. Для этой цели используется ключевое сло- во typedef.

Формат директивы:

typedef существующий\_тип псевдоним

Например, нужно описывать много указателей типа int\*, и нам хотелось бы, чтобы описание выглядело более наглядно, тогда можно определить синоним для типа и описать указатели примерно так:

typedef int\* IntPtr; IntPtr q, r, w;

Здесь переменные q,r,w — указатели типа int\*.

### Многомерные массивы

Массивы могут иметь и более одной размерности, описание *многомерных массивов* осуществляется следующим образом:

тип имя [размер 1][размер 2]...

В отличие от языка Паскаль, здесь для каждой размерности нужно писать свою пару скобок, например:

double q[3][4]; // Двумерный массив 3\*4 типа double

Индексы массива всегда отсчитываются от 0. При таком описании в памяти выделяется непрерывная область, причем правый индекс меняется быстрее левого.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| q[0][0] | q[0][1] | ... | q[0][3] | q[1][0] | q[1][1] | ... | q[2][3] |

Получить значение элемента массива при помощи указателя можно следую- щим образом:

double x;

x = \*q; // x = q[0][0]

x = \*(q + 2); // x = q[0][2]

x = \*(q + 4); // x = q[1][0]

Тип указателя служит для автоматического определения размера переменной, чтобы можно было правильно найти элемент массива в операции разадреса- ции \*(p + n).

Для двумерного массива переменная с одним индексом также является указа- телем и указывает на первый элемент строки матрицы, так:

q[0] - указатель на элемент q[0][0], q[1] - указатель на элемент q[1][0]. q[2] - указатель на элемент q[2][0].

Многомерные массивы можно при описании инициализировать, для этого, помимо общих скобок, каждая строка матрицы заключается в фигурные скобки, например:

double q[3][4] = { { 1, 0, 0, 0 },

{ 0, 1, 0, 0 },

{ 0, 0, 1, 0 }

};

### Динамическое выделение памяти

Часто возникает проблема выделения памяти для переменной уже в процессе работы программы (динамическое выделение). Если в классическом С эту задачу можно было решить, используя функцию malloc() или ее модифика- цию, то в С++ предусмотрен специальный оператор new. Если нужно выде- лить память под переменную, то операндом служит имя типа, а результатом работы оператора будет адрес памяти (указатель), расположенной в куче (ди- намической области памяти):

new тип;

Например:

int \*q = new int;

Если же необходимо выделить блок памяти (массив), его размер указывается после имени типа в квадратных скобках, а возвращаемым значением будет указатель на начало массива, например:

double\* v = new double[100];

Выделенная область памяти доступна, пока не будет освобождена операто- ром delete. После завершения работы программы выделенная в куче память освобождается.

Аргументом оператора delete будет указатель для простой переменной, или же указатель с пустыми квадратными скобками для выделенного массива, например:

delete q; delete[] v;

new в конечном итоге сводится к С-функции malloc().  Правило освобождения памяти простое: если в операторе new были квад-



Вообще-то оператор

ратные скобки — они должны быть и в операторе delete.

Причина специфического описания оператора delete состоит в том, что при выделении памяти массиву оператором new происходит выделение на два байта памяти больше, и в этих двух байтах, находящихся перед началом мас- сива, записан его размер в виде целого числа. Таким образом, наличие скобок в операторе delete[] является указанием, что размер массива записан на два байта ранее адреса v.

**Листинг 1.8. Пример программы с динамическим выделением памяти**

#include <iostream.h> void main()

{ int \*q, \*k = new int[10]; int i = 0;

while (i < 10) k[i++] = i;

for (q = k,i = 0; i < 10; i++) cout << \*q++ << endl; delete[] k;

}

В листинге 1.8 рассмотрен пример, демонстрирующий возможность обраще- ния к элементам массива как по индексу, так и по указателю при динамиче- ском выделении памяти. Причина, по которой мы вынуждены ввести еще один указатель, заключается в том, что при изменении указателя k мы не вправе использовать оператор освобождения памяти delete[] k;, поскольку в этом случае система будет пытаться освободить память по адресу, по кото- рому она не выделялась, что приведет к ошибке времени выполнения.

В листинге 1.9 приведем пример динамического выделения памяти для дву- мерного массива. В этом случае нам потребуется описать *указатель на ука-*

*затель* \*\*q, поскольку здесь \*q должен быть массивом указателей на строки массива.

**Листинг 1.9. Динамическое выделение памяти для двумерного массива**

#include <iostream.h> void main()

{ int i,j;

double \*\*q = new double \*[9];

for (i = 0; i < 9; i++) q[i] = new double[9]; for (i = 0; i < 9; i++)

{ for (j = 0; j < 9; j++)

cout << (q[i][j] = (i+1)\*(j+1)) << '\t'; cout << endl;

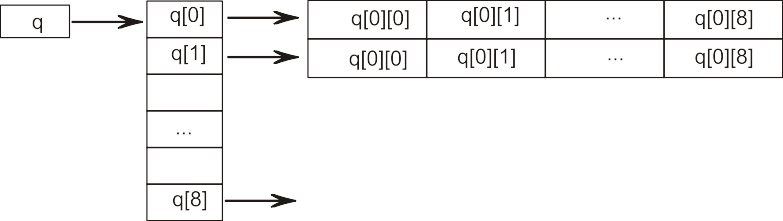
}

for (i = 0; i < 9; i++) delete[] q[i]; delete[] q;

}

Схема выделения памяти для двумерного массива приведена на рис. 1.6.





**Рис. 1.6.** Иллюстрация механизма выделения памяти для двумерного массива

Очевидно, что в этом случае освобождение памяти должно осуществляться в обратном порядке, т. к. вначале необходимо освободить память, выделенную для каждой строки массива, и лишь затем освободить память, выделенную под массив указателей на эти строки.



Здесь мы совместили вывод на консоль и вычисление элемента массива

cout << (q[i][j] = (i+1)\*(j+1)) << '\t';

воспользовавшись побочным эффектом — скобка возвращает результат по- следней выполненной операции.

#### Пример программы

**с использованием двумерного массива**

Рассмотрим в качестве примера задачу нахождения решения системы линей- ных уравнений методом Гаусса.

*A*11\**x*1 + *A*12\**x*2 + + *A*1*n*\**xn* = *B*1

*A*21\**x*1 + *A*22\**x*2 + + *A*2*n*\**xn* = *B*2

.....................................

*An*1\**x*1 + *An*2\**x*2 + + *Ann*\**xn* = *Bn*

Идея метода состоит в том, чтобы последовательно исключать неизвестные из уравнений с тем, чтобы последнее уравнение стало уравнением с одним неизвестным. Это прямой ход метода Гаусса. Первый его шаг состоит в том, чтобы исключить переменную *x*1 из второго и до n-го уравнений. Это можно сделать, умножив первое уравнение на коэффициент *A*21/*A*11, и вычесть его из второго уравнения. Аналогично поступим со следующими уравнениями, ис- пользуя коэффициент *Ai*1/*A*11. Нетрудно заметить, что при таком преобразова- нии коэффициенты при *x*1 в этих уравнениях обращаются в нуль. Повторяя процедуру *n*–1 раз и исключая переменные *x*2, *x*3, из следующих строк, мы

приведем систему к треугольной форме:

*A*11\**x*1 + *A*12\**x*2 + + *A*1*n*\**xn* = *B*1

*A*22\**x*2 + + *A*2*n*\**xn* = *B*2

...................

*Ann*\**xn* = *Bn*

Здесь коэффициенты *A*22 и последующие модифицированы и определяются на каждом шаге по формулам:

*Ajk* = *Ajk* – *Aik*\*(*Aji*/*Aii*), *Bj* = *Bj* – *Bi*\*(*Aji*/*Aii*).

Затем осуществляется обратный ход метода Гаусса, который заключается в том, что мы из последнего уравнения определяем *xn* и, подставляя его в пре- дыдущее уравнение, определяем *xn*–1, и т. д., пока не определим *x*1.

Реализуем данный алгоритм (листинг 1.10), где для повышения точности осуществим выбор главного элемента, т. е. найдем строку с максимальным значением коэффициента *Aii* и обменяем строки уравнения так, чтобы урав- нение с максимальным по модулю первым коэффициентом было верхним.

**Листинг 1.10. Программа решения системы линейных уравнений методом Гаусса**

#include <iostream.h> #include <string.h> #include <math.h> int main()

{ char \*p,str[82];

int i, j, k, n;

double a[10][11],x[10],q,z;

cout << "The decision of systems of the linear equations\n"

<< "Enter factors of system on lines\n";

// Ввод первой строки и определение размерности системы cin.getline(str,80); // Ввод первой строки

p = strtok(str," \t,;"); // Выделение первой лексемы i = 0;

do { a[0][i++] = atof(p);

p = strtok(NULL," \t,;"); // Выделение следующей лексемы

} while (p);

n = --i; // Размерность системы

// Ввод остальных строк

for (k = 1; k < n; k++)

{ cin.getline(str,80); // Ввод следующей строки p = strtok(str," \t,;");

i = 0;

do { a[k][i++] = atof(p);

p = strtok(NULL," \t,;");

} while (p);

}

// Прямой ход метода Гаусса

for (i = 0; i < n - 1; i++)

{ // Выбор главного элемента

//////////////////////////////////////////////////////////// z = fabs(a[i][i]);

for (j = i,k = i + 1;k < n;k++)

if ((q = fabs(a[k][i])) > z) z = q, j = k; if (j != i){ for (k = i;k <= n;k++) // Обмен строк

z = a[i][k], a[i][k] = a[j][k], a[j][k] = z;

*}*

//////////////////////////////////////////////////////////// for (j = i + 1;j < n;j++)

{ q = a[i][i];

if (fabs(q) < 1e-300)

{ cout << "The system is incompatible\n"; return 1; }

z = a[j][i]/q;

for (k = i + 1;k <= n;k++) a[j][k] -= a[i][k]\*z; }

} k = n - 1;

if (fabs(q = a[k][k]) < 1e-300)

{ cout << "The system is incompatible\n"; return 1; }

// Обратный ход метода Гаусса x[k] = a[k][n]/q;

for (i = k-1; i >= 0;i--)

{ z = 0.0;

for (j = k; j > i; j--) z += a[i][j]\*x[j];

x[i] = (a[i][n] - z)/a[i][i];

}

cout << "\nThe decision of system of the equations:\n";

for(i = 0;i<n;i++) cout << "x(" << i + 1 << ")=" << x[i] << endl; return 0;

}

#### Комментарии к программе

Для организации ввода матрицы коэффициентов системы уравнений по стро- кам опишем символьный массив char str[82]. Для хранения коэффициентов используем двумерный массив размерностью 10×11 double a[10][11]. Это означает, что мы можем решить систему из не более чем десяти уравнений. Свободные члены будем записывать в тех же строках, поэтому столбцов в матрице у нас 11.

Вводим первую строку коэффициентов в символьный массив при помощи метода оператора потокового ввода cin.getline(str,80), где указан массив, куда нужно ввести строку и максимальное количество символов. Мы не мо- жем в этой ситуации использовать оператор форматного ввода cin >> str, поскольку оператор вводит до первого разделителя, которым является знак пробела.

Для выделения числовых коэффициентов из введенной строки воспользуемся стандартной функцией strtok(), прототип которой находится в файле вклю- чений string.h. Функция принимает в качестве параметров исходную строку, которую нужно разделить на *лексемы* (фрагменты текста между разделителя- ми), а также строку, содержащую разделители в виде набора символов. Функция находит позицию первого разделителя и вместо него вставляет в исходную строку символ конца строки, после чего возвращает указатель на начало строки.

p = strtok(str," \t,;");

Мы задали в качестве разделителей символы пробела, табуляции, запятой и точки с запятой. Таким образом, при первом обращении мы получаем указа-

тель строки, содержащий всего одно число, которое может быть преобразо- вано к числу типа double при обращении к функции atof()из файла включе- ний math.h.

Однако для выделения следующей лексемы необходимо в функции strtok() вместо имени строки указать нулевой указатель NULL. Это является признаком для функции, что работа продолжается с той же строкой. Теперь функция пе- реставит признак конца строки на следующую лексему, а указатель возвратит также начала следующей лексемы. И так будет продолжаться до тех пор, пока не будет достигнут конец исходной строки, в этом случае функция strtok() возвращает нулевой указатель, что и будет служить признаком конца цикла:

do { ... } while (p);

По завершении обработки первой строки мы можем определить размер- ность системы — это на 1 меньше количества прочитанных коэффициентов: n = --i.

Сейчас, когда известна размерность системы, остальные строки ввести можно и проще, но мы, для единообразия процесса, поступим аналогично предыду- щему.

Теперь, после того как исходные данные введены, начинается прямой ход метода Гаусса. Организуем цикл по i, начиная с первой строки матрицы до предпоследней. Выбор главного элемента для начала можно и опустить, но в этом случае, если первый коэффициент окажется равным нулю, у нас могут возникнуть проблемы.

Внутри этого цикла организуем цикл по j для перебора всех последующих строк матрицы коэффициентов, причем, если коэффициент верхней строки a[i][i] равен нулю, мы считаем, что определитель системы равен нулю и система несовместна, в связи с чем дальнейшие вычисления не имеют смысла.

Здесь имеется проблема сравнения числа с плавающей точкой с нулем. Если просто написать a[i][i] == 0, это вряд ли будет хорошо, поскольку преобра- зование из символьного представления числа во внутреннее представление происходит приближенно и сохраняется лишь 16 значащих цифр, в связи с чем возникает вопрос, а 10–308 — это нуль или не нуль? Обычно в такой си- туации сравнивают с очень маленьким числом, как мы и поступаем, сравни- вая по модулю с константой 10–300. Если вспомнить нижний предел для чисел типа double, то это практически и есть нуль.

Теперь вычисляем коэффициент z = a[j][i]/q, который является общим для преобразуемой строки, в том числе и для правой части уравнения. Далее, еще в одном цикле, производим преобразование коэффициентов текущей строки:

for (k = i + 1;k <= n;k++) a[j][k] -= a[i][k]\*z;

На очередном шаге цикла по j преобразуем следующую строку и т. д. до по- следней строки. Затем, в цикле по i, спускаемся на строку ниже и повторяем процедуру, пока не дойдем до последней строки матрицы.

Прямой ход метода Гаусса завершен. Матрица коэффициентов сейчас имеет треугольную форму и, если последний коэффициент отличен от нуля, можно приступить к реализации обратного хода метода Гаусса — найти x[k] из по- следнего уравнения, которое является уравнением с одним неизвестным:

x[k] = a[k][n]/a[k][k];

После чего, полученное решение подставляется в предыдущее уравнение и находится x[k-1] и т. д., пока мы не доберемся до первой строки и не найдем x[0].

Задача решена, осталось лишь вывести результаты на консоль.

## Функции

Классический С — язык сугубо процедурный и основной логической едини- цей программы является *функция*. Формат описания функции следующий:

[тип возвращаемого значения] имя\_функции(список параметров)

{

тело функции

[return возвращаемое\_значение]

}



В скобках помещена необязательная часть конструкции. Вообще, функция должна возвращать какое-то значение, но если указать тип возвращаемого зна- чения void, то функция ничего не возвращает и оператор return не нужен. Если же совсем не указывать тип возвращаемого значения, то по умолчанию компилятором понимается тип int.

Имя функции образуется по тем же правилам, что и имя переменной, и жела- тельно, чтобы имя отражало назначение функции.

Список параметров:

тип\_1 имя1[, тип\_2 имя2, ... тип\_n имя\_n = const\_val]

В отличие, например, от языка Паскаль, тип указывается для каждой пере- менной. Также допускается задание значения по умолчанию для последних в списке параметров.

Все функции, используемые программой, должны быть предварительно опи- саны. Если это пользовательские функции, то их описание можно поместить

перед головным модулем main(), или же поместить перед ним *прототип* функции, а само описание вынести в конец файла. Прототип функции состо- ит из ее заголовка, после которого вместо тела функции стоит точка с запя- той, он необходим компилятору для того, чтобы правильно установить тип и количество параметров, а также тип возвращаемого значения. Имена пара- метров в прототипе можно опустить, поскольку компилятор все равно их иг- норирует, так, например, можно написать:

int min(int, int);

Если же мы используем в программе стандартные библиотечные функции, то достаточно описать в начале программы соответствующий файл включений, как раз и содержащий описание прототипов этих функций. А тело библио- течной функции расположено в откомпилированном виде в соответствующем библиотечном файле, доступ к которому получит программа-компоновщик при создании исполняемого модуля.

Рассмотрим, как можно было бы написать функцию вычисления минималь- ного значения двух переменных типа int:

int min(int a, int b)

{ return (a < b)? a : b;}

Здесь мы воспользовались арифметическим условным оператором. Или еще один пример, обмен местами двух переменных типа int:

void swap(int \*a, int \*b)

{ int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

Функция ничего не возвращает, но, поскольку принимает в качестве парамет- ров указатели (по сути, адреса переменных), то имеет возможность работать со значениями этих переменных и поменять их местами. Для извлечения зна- чения переменной используется операция разадресации \*a;.

Обращение к функции:

int k = 1, m =2; swap(&k, &m);

При использовании функции нужно передавать ей либо указатели на пере- менные, либо использовать операцию взятия адреса &.

В следующем листинге (листинг 1.11) приведен пример сортировки "методом пузырька". *Сортировка* означает упорядочение, в нашем случае — по возрас- танию.

**Листинг 1.11. Сортировка "методом пузырька"**

#include <iostream.h>

void sort(int n, double x[])

{ int i,j; double z;

for (i = 0; i < n-1; i++)

for (j = n-1; j > i; j--)

if (x[j] < x[j-1]) z = x[j], x[j] = x[j-1], x[j-1] = z;

}

void main()

{

double x[] = {1.2, -8, 3.4, 7,8, 0, 8.1, 10};

int k = sizeof(x)/sizeof(double); sort(k, x);

for (int i = 0; i < k; i++) cout << x[i] << endl;

}

Функция принимает в качестве параметров количество элементов массива и указатель начала массива данных. Просматривает, начиная с конца массива каждую пару значений, и обменивает их местами, если меньшее число лежит ниже. Меньшие числа как бы "всплывают" наверх, как пузырьки воздуха, от- сюда и название метода. Таким образом, просмотрев весь массив, мы пере- мещаем самое маленькое число наверх, после этого, начиная со следующего элемента массива, повторяем процедуру. Проделав это *n*–1 раз, мы пол- ностью упорядочим массив.

### Встраиваемые функции

Накладные расходы по вызову функции достаточно велики, поэтому для не- больших функций, состоящих из одного-двух операторов, имеется возмож- ность объявить их *встраиваемыми* при помощи ключевого слова inline. Это будет служить рекомендацией компилятору — вместо построения функции встроить ее код. Если компилятор сочтет, что функция слишком велика, он проигнорирует это указание.

Например, функцию, вычисляющую минимальное значение двух перемен- ных, можно было бы построить так:

inline int min(int a, int b)

{ return (a < b)? a : b;}

### Указатели на функции

Можно определить *указатель на функцию*. Например, функция, которая вы- зывается без аргументов и не имеет возвращаемого значения, может быть оп- ределена следующим образом:

void (\*func)();

Присваивание значения указателю осуществляется так:

func = f;

где f — функция, описанная как:

void f() { ... }

Нетрудно, по аналогии, построить указатель на функцию, принимающую пе- ременные типа int и double, а возвращающую значение типа double:

double (\*fd)(int, double);

Можно также описать и массивы указателей на функции, например:

void (\*func[10])();

### Примеры некоторых функций работы со строками

Эти и множество других функций описаны в файле включений string.h, но мы рассмотрим их организацию, чтобы лучше понять принципы их функцио- нирования.

#### Вычисление длины строки

int strlen(char \*s)

{ for (int i = 0; \*s++; i++); return i;

}

Функция принимает указатель на строку, мы используем его для чтения сим- волов операцией разадресации \*s, что возвращает нам текущий символ, кото- рый есть true, если он не является завершающим символом строки, после че- го указатель увеличивается s++. Таким образом, мы доберемся до конца строки '\0', а поскольку мы наращивали параллельно индекс i++, то при вы- ходе из цикла переменная i равна длине строки. Ее и возвращает функция. Тело цикла заменяет *пустой оператор* (;). Обращение же к этой функции будет выглядеть так: n = strlen(str);.

#### Заполнение строки

void strfill(char \*s, int n, char ch = ' ')

{ for (int i = 0; i < n; i++) \*s++ = ch;

\*s = '\0';

}

Эта функция принимает три параметра, и если первые два не вызывают во- просов, то последний, char ch = ' ', требует некоторых комментариев. В расширении С++ допускается при описании параметров функции задавать значение по умолчанию. Можно обратиться к функции с двумя парамет- рами, тогда символ-заполнитель будет равен значению по умолчанию strfill(str,10). Если же обратиться к функции с тремя параметрами, то мы можем изменить значение символа-заполнителя, например strfill(str,10,'\*'). Отметим только, что параметры функции с определен- ным значением по умолчанию должны быть последними в списке парамет- ров. В цикле for присваиваем n символам массива значение символа- заполнителя, после чего самостоятельно формируем признак конца текстовой строки \*s = '\0'. Здесь указатель s как раз и указывает на символ после последнего заполненного. Возвращаемого значения функции не требуется, поскольку при передаче в функцию в качестве параметра указателя операция заполнения осуществляется с исходной строкой.



Типичная ошибка при формировании строки как раз и заключается в том, что забывают про завершающий нулевой символ.

#### Копирование С-строк

void strcpy(char \*out, char \*in)

{

while(\*out++ = \*in++);

}

Очень элегантное решение задачи с использованием указателей. Посимволь- но копируем содержимое исходной строки с автоматическим увеличением обеих указателей. При копировании завершающего нулевого символа цикл завершается. Тело цикла пустое, возвращаемого значения мы также не преду- сматриваем.

#### Обращение строки

char\* revers(char s[])

{ int i,j,n = strlen(s); char ch, \*p, \*q;

for (p = s, q = s + (n-1); p < q; p++, q--)

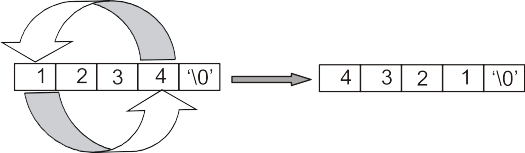
{ ch = \*p; \*p = \*q; \*q = ch; } return s;

}



Используем здесь функцию strlen(), описанную ранее.

Эту задачу можно решить несколькими способами, но наиболее эффектив- ный код, как представляется, будет с использованием указателей. Мы опре- делим два указателя и установим первый на начало, а второй — на конец строки. Теперь обмениваем символы в цикле, пока второй указатель больше первого (рис. 1.7).



**Рис. 1.7.** Иллюстрация к функции revers()

Здесь мы еще поставили оператор return для возврата указателя на строку. Сейчас можно использовать эту функцию и в потоке вывода, что иногда бы- вает удобно, например:

cout << revers(str);

## Ссылки

*Ссылочный тип данных* введен в С++ как расширение синтаксиса классиче- ского языка С. Ссылка объявляется со знаком амперсанда &, является просто псевдонимом имени переменной и должна быть связана с существующей пе- ременной сразу при описании.



За исключением передачи ссылки как параметра функции.

Пример:

int k; int& q = k;

Теперь можно использовать переменные k и q как синонимы.

k = 1; cout << q; // Будет выведена 1

На самом деле, при объявлении ссылочной переменной компилятор не созда- ет отдельной переменной, а работает с исходной переменной, поэтому ссылке не может быть присвоено другое значение, также не разрешается создавать массивы ссылок. Наиболее удобно использование ссылочных переменных при передаче параметров в функцию вместо указателей.

Рассмотрим в качестве примера, как можно переопределить функцию swap()

с использованием ссылок:

void swap(int& a, int& b)

{ int tmp = a;

a = b; b = tmp;

}

Очевидно, такое определение выглядит более ясно. Здесь переменные a и b передаются по ссылке, поэтому в тексте функции под a и b понимаются именно те переменные, которые являются аргументами функций. Обращение к функции сейчас будет выглядеть так:

swap(k,m);

### Примеры простых вычислительных задач

#### Решение трансцендентного уравнения

Рассмотрим в качестве примера реализацию простейшего алгоритма решения методом половинного деления трансцендентного уравнения или, другими словами, уравнения, не имеющего решения в элементарных функциях.

x = cos(x)

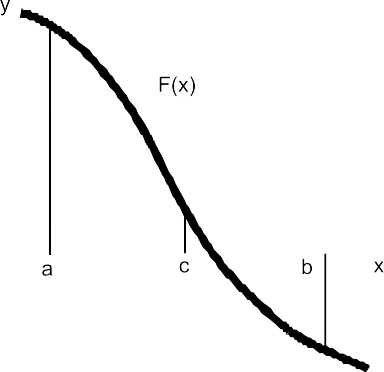
Суть метода заключается в следующем. Составляем функцию (рис. 1.8):

F(x) = cos(x) - x.

Очевидно, что решение уравнения есть нуль этой функции, поэтому сформу- лируем задачу как поиск нуля функции F(x). Решение задачи представлено в листинге 1.12.

Алгоритм можно описать так:

1. Вводим интервал для поиска решения a, b.
2. Начало цикла.
3. Делим интервал пополам, получаем точку c = (a + b)/2.



**Рис. 1.8.** Иллюстрация алгоритма метода половинного деления

1. Если знаки функции F(x) в точках a и c различны, решение находится на интервале (a,c), тогда положим b = c;, иначе, если знаки функции в точ- ках c и b различны, решение находится на интервале (с,b), тогда положим a = c, иначе прекращаем работу, поскольку решения нет.
2. Пока b-a > e, возвращаемся к началу цикла, иначе выводим середину ин- тервала (a + b)/2 и заканчиваем работу.

**Листинг 1.12. Программа решения трансцендентного уравнения методом половинного деления**

#include <iostream.h> #include <math.h> const double e = 1e-6;

double f(double x) // Вычисление значения функции

{ return cos(x) - x; } int main()

{

double a,b,c;

cout << "Method of halving\n";

cout << "Enter borders of an interval [a, b]: "; cin >> a >> b;

if ( a > b) c = a, a = b, b = c; // "Защита от дурака" do

{ c = (a + b)/2.0;

if (f(a)\*f(c) < 0) b = c;

else if (f(c)\*f(b) < 0) a = c;

else{ cout << "Decisions in the given interval\ not exist\n"; return 1; }

}

while (b - a > e);

cout << "x = " << (a + b)/2.0 << endl; return 0;

}

#### Некоторые комментарии к программе

1. Оператор return 1; обеспечивает завершение работы программы в случае задания неверного интервала поиска решения и возвращает 1. Для опера- ционной системы любой ненулевой код принято считать ненормальным завершением работы программы. В этом случае мы должны описать тип возвращаемого значения головной функции int main().
2. Константу, задающую точность вычислений, мы определяем как const double e = 1e-6;.
3. Поскольку функцию мы описываем перед ее употреблением в функции

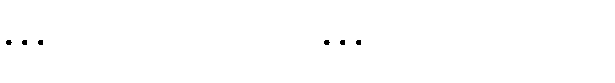
main(), то нет необходимости описывать ее прототип.

1. После ввода границ интервала мы используем так называемую "защиту от дурака", т. е. предусматриваем ситуацию, когда пользователь введет ин- тервал наоборот.
2. Смену знака функции на отрезке мы проверяем по знаку произведения значений функции на границах (f(a)\*f(c) < 0). Решение не самое эффек- тивное, но простое.
3. И наконец, в строковой константе "Decisions in the given interval\, ко- торая не поместилась в строку, мы использовали соглашение языка С++ о переносах, а именно в конце строки ставится символ \ "обратный слэш", а продолжение — в следующей строке программы not exist\n";.

#### Суммирование бесконечных рядов

Рассмотрим задачу о суммировании бесконечного ряда на примере широко известного разложения:

sin *x*



3 5

*x x x*

3! 5!

1 *n x*2*n* 1

2*n* 1 !

.

Однако вычислять сумму прямо по этой формуле вряд ли целесообразно, по- скольку при *x* > 1 числитель дроби быстро растет, знаменатель также быстро- растущая функция, а при делении одного большого числа на другое большое число ошибки округления начинают сильно сказываться. К тому же, мы очень быстро придем к переполнению в знаменателе. Поэтому для решения подобной задачи лучше подобрать рекуррентное соотношение, т. е. найти

формулу, связывающую очередное слагаемое с предыдущим. Нетрудно убе- диться, что следующее выражение и будет искомым рекуррентным соотно- шением:

*sn*



*x*2 *sn* 1 2*n* 2*n*

.

1

Построим программу табулирования суммы этого ряда на отрезке [0; /2] (листинг 1.13), пользуясь полученной формулой. Причем оборвем суммиро- вание тогда, когда очередное слагаемое станет меньше заданной точности. Поскольку ряд знакопеременный, то точность будет достигнута (согласно теореме об оценке остаточного члена в знакопеременном ряде).

|  |
| --- |
| **Листинг 1.13. Программа суммирования бесконечного ряда** |

#include <iostream.h> #include <math.h> main()

{ const double PI\_2 = 1.571, E = 1e-8; double x, sum, sn, n, h = 0.1;

for (x = 0.0; x < PI\_2; x += h)

{

n = 1.0;

sum = sn = x; do {

sn \*= -x\*x/(2.0\*n\*(2.0\*n + 1.0)); sum += sn;

n += 1.0;

}

while (fabs(sn) > E);

cout << x << '\t' << sum << endl;

}

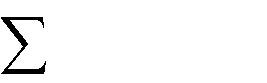
}

#### Комментарии к программе

1. Директивы включения обеспечивают доступ к оператору потокового вы- вода <<, а также математической библиотеке math.h, где нам потребуется функция fabs().
2. Константы, необходимые для работы программы, опишем как const double.
3. В цикле по x присваиваем начальные значения и начинаем суммировать члены ряда, проверяя, не достигли ли заданной точности, и наращиваем номер n. Здесь следует обратить внимание, что, несмотря на то, что значе- ние переменной n — целое, мы объявили ее как double, чтобы избежать неявного преобразования типа в каждой арифметической операции. Из этих же соображений целые константы 0, 1 и 2 мы описываем с десятич- ной точкой 0.0, 1.0, 2.0, чтобы компилятор построил их типа double.

#### Вычисление интеграла методом Симпсона

Рассмотрим вычисление определенного интеграла методом Симпсона (лис- тинг 1.14) по следующей формуле:



*n* 1

*h* 2

3

*f* (*x* ) 4 *f* (*x*

2*i*

2*i* 1

) *f* (*x*

2*i* 2

) ,

*i* 0

*S*

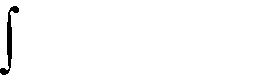
где *n* — количество интервалов разбиения области интегрирования, *h* — шаг разбиения (рис. 1.9).



**Рис. 1.9.** Разбиение интервала интегрирования

Вычислим для примера интеграл Френеля, который, как известно, не вычис- ляется в элементарных функциях.

1



sin(*x*2 )*dx* .

*F*

0

Табличное значение его известно, и мы сможем проконтролировать правиль- ность работы программы.

**Листинг 1.14. Программа вычисления определенного интеграла методом Симпсона**

#include <iostream.h> #include <math.h>

double f(double x) { return sin(x\*x); } main()

{ const double a = 0.0, b = 1.0, e = 1e-8; double s, w, x, h, n = 10.0;

cout << "Frenel Integral: "; s = 0.0;

do

{ w = s; h = (b - a)/n;

for (s = 0.0, x = a; x < b - h; x += 2.0\*h) s += f(x) + 4.0\*f(x + h) + f(x + 2.0\*h);

s \*= h/3.0; n \*= 2.0;

}

while (fabs(s - w) > e); cout << s << endl;

}

В результате должны получить значение *S* = 0.310268.

#### Комментарии к программе

1. В директивах включения мы подключили необходимые файлы, нам по- требуются оператор вывода и функция sin().
2. Далее описываем функцию double f(double x) { return sin(x\*x); }. По- скольку мы помещаем ее перед головной функцией, нет необходимости описывать прототип.
3. Основная проблема в вычислении интеграла заключается в решении во- проса — когда нужно прекратить дробление отрезка для достижения не- обходимой точности? Очевидно, что чем меньше интервал разбиения, тем точнее сумма криволинейных трапеций совпадает со значением интеграла. Мы поступаем просто, сравнивая два последовательно вычисленных зна- чения этой суммы при уменьшении величины интервала вдвое. Согласно теореме Рунге, точность вычисления, по меньшей мере, на порядок мень- ше модуля этой разности.

## Структуры

*Структура* является составной переменной и предназначена, в основном, для того, чтобы можно было оперировать с блоком разнотипных данных как с единым объектом. Формат объявления структуры следующий:

struct имя {

тип1 имя переменной 1; тип2 имя переменной 2;

...

};



Можно и сразу описать переменные структурного типа при описании структуры после закрывающейся скобки и перед точкой с запятой.

Структура определяет новый тип данных и описывается, как правило, вне описания функции, а переменные описываются, используя имя структуры как имя нового типа. Точка с запятой в конце описания структуры должна обяза- тельно присутствовать.

В качестве примера опишем структуру Student с двумя полями: name, number. struct Student

{ char name[20]; int number;

};

void main()

{ Student st;

...

}

Доступ к элементам структуры осуществляется посредством операции *"точ- ка"* над структурной переменной, например:

st.name – поле name переменной st. st.number – поле number переменной st.

#### Возможности структур

1. При описании структурной переменной возможна начальная инициализа- ция ее полей. Для этого начальные значения соответствующих типов запи- сываются в фигурных скобках аналогично инициализации массива. На- пример, для структуры Student:

Student st = {"P.Norton",200083 };

1. Присваивание структурных переменных.

В языке реализована возможность прямого присваивания структурных пе- ременных. В этом случае происходит просто копирование области памяти, занимаемой переменной-источником, в переменную-приемник. Например:

st1 = st;

1. Доступ к элементам структур через указатели.

Можно описать указатель на структурную переменную. Если теперь уста- новить этот указатель на переменную типа структуры, то доступ к элемен- там структуры осуществляется операцией ->. Операция описывается дву- мя символами: *"минус*" и *"больше"*. Например:

Student \*p = &st. p->name; // Имя

p->number; // Номер



Операцию -> следует называть *селектором выбора*.

Приведем пример программы с использованием структуры в листин- ге 1.15.

**Листинг 1.15. Программа, в которой используется структура**

#include <iostream.h> struct Student

{ char name[20]; int number;

};

void main()

{ Student st = {"P.Norton",200083 }; Student \*p, st1;

st1 = st; p = &st1;

cout << p->name << '\t' << p->number << endl;

}

В этом примере видно еще одно принципиальное отличие структуры от массива, а именно: если имя массива — это указатель на начало массива, то имя структуры — это сама структура, что позволяет осуществлять опе- рации присваивания, а при присваивании значения указателю структурной переменной требуется операция "взятия адреса" p = &st1;.

1. Возможность описания указателя структуры внутри структуры.

Часто возникает проблема при описании структуры описать там указатель на эту же структуру, и хотя описание структуры еще не завершено, фор- мат языка это позволяет. Например, опишем структуру List, представ- ляющую собой однонаправленный список, где в качестве данных опишем переменную типа int и указатель на эту же структуру List (лис- тинг 1.16).

**Листинг 1.16. Программа, создающая список и выводящая его содержимое на консоль**

#include <iostream.h> struct List

{ int dat; List\* next;

};

void main()

{ int i;

List \*p, \*heap = new List; // Начало списка for (p = heap, i = 0; i < 10; i++)

{ p->dat = i;

p = p->next = new List; // Порождаем следующий элемент списка

}

for (p = heap, i = 0; i < 10; i++)

{

cout << p->dat << endl;

p = p->next; // Читаем указатель следующего элемента списка

}

}

Здесь мы описали два указателя: heap — для указания начала списка, p — для передвижения по списку; и простую переменную, как счетчик цикла. В отли- чие от массива, наш список будет "разбросан" по памяти, поскольку оператор new выделяет первые свободные блоки памяти, но это неважно, поскольку мы передвигаемся по списку, используя сохраненные в самом списке адреса: p = p->next;.

## Объединения

*Объединения* похожи на структуры, но выполняют несколько иные функции. В объединении все переменные начинаются с одного адреса, они совмещены в памяти, что позволяет интерпретировать одну и ту же область памяти, как

данные разного типа. Размер объединения определяется максимальным раз- мером переменной.

Формат объединения отличается от структуры только служебным словом

union:

union имя {

тип1 имя переменной 1; тип2 имя переменной 2;

...

};

Объединение, как и структура, определяет новый тип данных и описывается, как правило, вне описания функции, а переменные описываются, используя его имя как имя нового типа. Точка с запятой в конце описания объединения должна обязательно присутствовать.

Доступ к элементам объединения осуществляется так же, как к элементам структуры — через точку для имени объединения, или по селектору, для об- ращения через указатель.

Например, имеется 4 флага, и мы хотели бы сократить время для операций с несколькими флагами сразу. Для простоты рассмотрим, как можно обну- лить все флаги одной операцией (листинг 1.17).

**Листинг 1.17. Пример программы с объединением**

#include <iostream.h> union Flag

{ long g; char ch[4];

};

void main()

{

Flag fl; fl.ch[0] = 1;

fl.ch[1] = 2;

fl.ch[2] = 4;

fl.ch[3] = 8;

cout << "Flag = " << hex << fl.g << endl;

fl.g = 0; // Все флаги равны 0 cout << "Flag = NULL\n";

for (int i = 0; i < 4; i++) cout << (int)fl.ch[i] << endl;

}

Мы описали объединение, как переменную типа long, и, одновременно, в виде массива типа char. Теперь для переменной типа Flag мы можем рабо-

тать с каждым флагом независимо, или, используя операцию с переменной типа long, обнулить все флаги простой операцией присваивания fl.g = 0;.

Здесь нужно пояснить некоторые особенности использования операторов вы- вода. Так, присутствие в потоке вывода конструкции << hex означает вызов так называемого манипулятора для вывода последующего числа в шестнад- цатеричной форме, чтобы мы могли увидеть значение отдельных флагов, действительно получив число 8040201, мы сразу видим каждый байт — это двузначное шестнадцатеричное число:

01 02 04 8.

В следующем операторе вывода мы хотели вывести каждый символ в виде целого числа, поэтому мы поставили оператор явного преобразования типа: (int)fl.ch[i].

## Описание головного модуля

Описание функции main() допускает три формы:

1. main() без параметров.
2. main(int argc,char \*argv[]), где:

 argc — количество переданных параметров;

 argv[0] — имя программы;

 argv[1] — первый параметр командной строки;

...



argv[n] — n-й параметр.

1. main(int argc, char \*argv[], char \*envp[]),

где envp[] — массив указателей на переменные окружения.

При запуске программы из командной строки параметры разделены пробе- лом, причем первым параметром по умолчанию считается имя запускаемого файла (пример такой программы см. в листинге 1.18).

**Листинг 1.18. Пример программы, которая запускается с параметрами в командной строке и выводит эти параметры на экран монитора**

#include <iostream.h> main(int n, char \*arg[])

{

for (int i = 0; i < n; i++) cout << arg[i] << endl;

}

Запуск программы c:\work>test a b c. Вывод:

test.exe a

b c

Можно просмотреть *переменные среды окружения*, учитывая, что массив указателей заканчивается нулевым значением (листинг 1.19).

|  |
| --- |
| **Листинг 1.19. Программа вывода переменных среды окружения** |

#include <iostream.h>

main(int n, char \*arg[], char \*env[])

{

cout << "Environment:\n";

for (int i = 0; env[i]; i++) cout << env[i] << endl;

}

Вывод:

Environment:

ALLUSERSPROFILE=C:\Documents and Settings\All Users APPDATA=C:\Documents and Settings\Application Data CLIENTNAME=Console

CommonProgramFiles=C:\Program Files\Common Files COMPUTERNAME=MASTER

ComSpec=C:\WINDOWS\system32\cmd.exe HOMEDRIVE=C:

HOMEPATH=\Documents and Settings\ include=

INTDIR=

lib= LOGONSERVER=\\MASTER NUMBER\_OF\_PROCESSORS=1

OS=Windows\_NT OUTDIR=

Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem; PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH PROCESSOR\_ARCHITECTURE=x86

...

## Вопросы к главе

* 1. Назначение директив препроцессора #include, #define.
  2. Этапы создания проекта консольного приложения в среде Visual Stu- dio 6.0.
  3. Основные типы данных классического языка С, новые типы данных С++. Диапазон значений, размер типа.
  4. Запись констант в языке С++. Управляющие символьные константы. Особенности записи строковых констант.
  5. Формат описания переменных, инициализация переменных, константные переменные.
  6. Классы памяти.
  7. В чем своеобразие операции присваивания языка С++?
  8. Арифметические операции, приоритеты операций.
  9. Правила неявного преобразования типов. Форматы явного преобразова- ния типов.
  10. Операции ++ и --. Префиксное и постфиксное использование.
  11. Битовые операции, их использование для преобразования данных.
  12. Комбинированные операции.
  13. Операции отношения.
  14. Логические выражения и операторы.
  15. Оператор перехода goto, целесообразность его использования.
  16. Условный оператор if ... else. Сокращенная и полная форма записи оператора.
  17. Условный арифметический оператор ()?.
  18. Три формы операторов цикла. Вспомогательные операторы break, continue.
  19. Переключатель switch.
  20. Описание одномерного массива и массивов большей размерности. Ини- циализация массива.
  21. Указатели, их описание. Имя массива — указатель? Операция "взятия адреса", разадресация указателя. Арифметика указателей.
  22. Как определить псевдоним типа данных?
  23. Операторы выделения и освобождения памяти для переменной и массива.
  24. Ссылочный тип данных. Связь с указателями.
  25. Структура — пользовательский тип данных. Доступ к элементам струк- туры по имени и через указатель. Операция присваивания для структур.
  26. Объединение, его назначение.
  27. Механизм передачи параметров головному модулю С++-программы.

## Задание для самостоятельной работы

1. Напишите программу, реализующую арифметические операции с целыми типами данных. Посмотрите, как работает операция вычисления остатка от деления "%" с отрицательными числами.
2. Напишите программу, в которой умножение и деление целого числа на 2n

реализуется при помощи логических сдвигов.

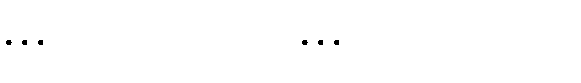
1. Найти порядок n для целого числа k < 2n при помощи операций сдвига.
2. Реализуйте программу решения квадратного уравнения с обязательной проверкой существования действительных корней, а также кратного кор- ня. Функцию вычисления квадратного корня sqrt() найдите в файле включений math.h.
3. Составьте программу табулирования функций: sin(x), cos(x), x2, на интервале [0, /2].



x

1. Определите одномерный массив из 10 чисел с плавающей точкой в тексте программы. Вычислите минимальное и максимальное значение и пози- ции этих чисел в массиве, сумму элементов массива и среднее значение.
2. Решите предыдущую задачу с вводом 10 чисел с консоли.
3. Определите двумерный массив размером 4 4 из целых чисел в тексте программы. Вычислите минимальное значение в каждой строке, сумму элементов массива в каждом столбце. Найдите сумму и произведение элементов главной и побочной диагонали.
4. Решите предыдущую задачу, вводя матрицу значений с консоли.
5. Написать программу, которая вводит целое число n и выделяет память для массива из n данных типа char, int, double. Перед завершением рабо- ты программы память освободить.
6. Написать программу, которая вводит строку текста, разделяет ее на слова и выводит их на консоль по слову на строку. Разделителем слов считать пробел и знаки пунктуации, а для выделения слов использовать функцию strtok().
7. Написать функцию left(), принимающую количество символов, тексто- вую строку и выделяющую n символов в начале строки. Функция должна выделить память и вернуть указатель на возвращаемую строку.
8. Аналогично, написать текст функции right(), выделяющей n символов в конце строки.
9. Написать текст функции strcat(), "склеивающей" две текстовые строки. Функция принимает указатели на две строки s1 и s2, выделяет память для "склеенной" строки и возвращает ее указатель.
10. Написать текст функции strchr(), которая ищет первое появление в строке заданного символа. Функция возвращает указатель на найденный символ или нулевой указатель, если символ не найден.
11. Рассмотрите реализацию метода половинного деления для решения трансцендентного уравнения и напишите программу, реализующую для этой цели метод Ньютона.
12. Рассмотрите приведенный в тексте пример суммирования бесконечного ряда и, по аналогии, напишите программу табулирования с точностью 10–8 на отрезке [0; /2] функции:

cos(*x*)



2 4

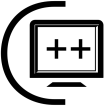
1 *x x*

2! 4!

1 *n x*2*n*

2*n* !

1. Создать структуру COMPLEX для записи комплексных чисел. Реализовать арифметические операции с комплексными числами: +, -, \*, /. Написать функцию print() для вывода на консоль комплексного числа.
2. Рассмотреть пример однонаправленного списка, построенного на струк- туре List. Измените структуру так, чтобы получить возможность пере- двигаться по списку не только вперед, но и назад (двунаправленный список).
3. Напишите программу, которая выводит на консоль значение переменной окружения "OS", используя формат функции main() с тремя параметрами.

**Г ЛА В А 2**

# Классы

Принципиальное отличие языка программирования С++ от классического С заключается в наличии в последнем классов. Идея введения классов заключа- лась в том, чтобы объединить данные и те функции, которые с этими данны- ми работают. Функции в данном контексте будут называться *методами*. Мы уже рассматривали структуры, которые объединяют данные различных типов в один объект, так вот, если сюда добавить и функции, которые в данном контексте будут называться методами, то мы получим *класс*.

Синтаксис описания класса:

class имя

{

private: определение переменных и закрытых членов класса

public: определение открытых членов класса

};

Введение классов существенно изменило стиль программирования, и если классический язык С был языком процедурным, то С++ стал языком объект- но-ориентированным. В связи с этим появилось желание вообще скрыть пе- ременные класса от внешнего мира и позволить работать с ними напрямую только членам своего класса, что называется *инкапсуляцией*, или сокрытием данных. Механизм реализован введением служебного слова private, которое определяет все, что стоит после него, как закрытые члены класса. И, соответ- ственно, служебное слово public определяет открытые члены класса. По умолчанию все члены класса определяются как закрытые.

Вообще-то определить функции можно и в структуре. Практически все, что мы будем описывать как свойства класса, относится и к структуре, а единст- венным принципиальным отличием будет то, что все члены структуры, по умолчанию, объявляются как открытые, в то время как все члены класса, по умолчанию, закрытые.

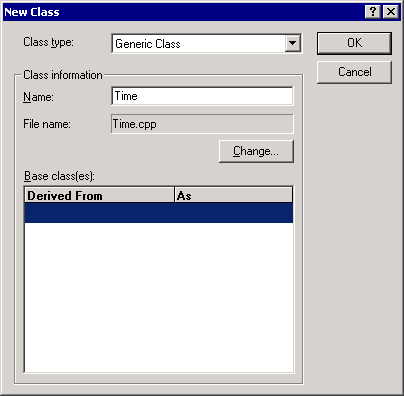
## Создание нового класса в среде Visual C++ 6.0

Рассмотрим технику создания класса на примере простого класса Time.

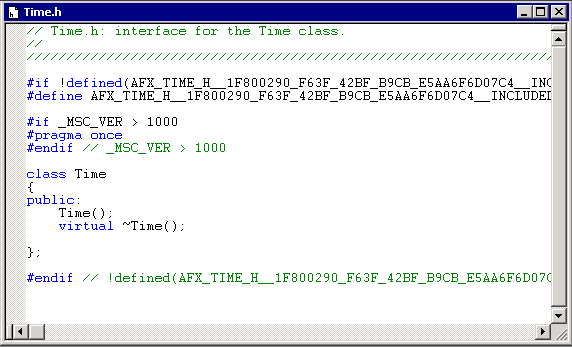
1. Точно так же, как для простого приложения, создадим пустой проект для 32-битного консольного приложения (**Win32 Console Application**).
2. Добавим в проект файл для описания головной функции **C++ Source File**, зададим его имя, например, main. (Расширение имени файла .cpp добавится автоматически.)
3. В окне **Workspace** перейдем на вкладку **ClassView**. Установив курсор мыши на имя **Time classes**, выберем в контекстном меню пункт **New Class**. Введем имя класса Тime (рис. 2.1), после чего будут автоматически соз- даны заготовки для класса: файл Time.h и файл реализации Time.cpp (рис. 2.2).
4. Будем описывать класс сразу в заголовочном файле, поэтому файл реали- зации Time.cpp просто удалим из проекта (выделить и нажать клавишу

<Delete>).

1. В заголовочном файле Time.h приведены прототипы для конструктора и деструктора. Деструктор (его имя ~Time) пока удалим. В листинге 2.1 приведен текст файла Time.h для дальнейшей работы.



**Рис. 2.1.** Окно создания нового класса



**Рис. 2.2.** Заготовка файла Time.h

**Листинг 2.1. Текст файла Time.h с описанием класса работы со временем Time**

#include <iostream.h> #include <iomanip.h> class Time

{private: int hours; // Часы int minutes; // Минуты

public:

}

Time(): hours(0), minutes(0) { }

Time(int h, int m = 0): hours(h), minutes(m) { } Time(Time& t): hours(t.hours), minutes(t.minutes) { } int gethours() { return hours;}

int getminutes() { return minutes; } void set(int h = 0, int m = 0)

{ hours = h; minutes = m; } void addmin(int m)

{ minutes += m; hours += minutes/60; minutes %= 60;

void addhours(int h){ hours += h; } void show()

{ cout<<setw(2)<<hours<<':'<<setw(2)<<minutes<<endl;} Time operator + (Time& t)

{ Time tmp;

tmp.minutes = minutes + t.minutes;

tmp.hours = hours + t.hours + tmp.minutes/60; tmp.minutes %= 60;

return tmp;

}

};

Заполним файл main.cpp следующим текстом с описанием головной функции (листинг 2.2).

**Листинг 2.2. Файл, содержащий головную функцию программы**

#include "Time.h" // Подключение файла с описанием класса void main()

{

Time t, tf, tw(8); // Создание объектов конструктором cout << setfill('0'); // Шаблон заполнения

tw.show(); // Вызов метода tw.addmin(90);

Time k = tw; // Работает конструктор копирования k.show();

t.addhours(-4);

t.show();

tw.set(1,3);

tf.set(5);

t = tw + tf; // Перегруженный оператор суммирования t.show();

}

### Подробное описание класса *Time*

Вначале описываем две переменных целого типа, как закрытые члены класса.

private: int hours; // Часы

int minutes; // Минуты

Затем, в открытой части класса, описываем *конструктор* по умолчанию:

Time(): hours(0), minutes(0) { }



Важно то, что конструктор описывается именно в открытой части класса, в про- тивном случае мы не смогли бы создать ни одного объекта этого класса.

Задача этого конструктора заключается в присваивании начального значения переменным. Только после описания конструктора по умолчанию мы можем описывать в программе переменные без задания начальных значений:

Time t;

Одной из особенностей языка С++ является то, что при создании любого объ- екта компилятор автоматически использует конструктор, что позволяет из- бежать проблем, связанных с начальной инициализацией переменных. Более того, если мы не опишем в классе ни одного конструктора, то компилятор сам построит конструктор по умолчанию, правда делать он ничего не будет.

Здесь нужно еще отметить довольно своеобразный для классического С спо- соб инициализации переменных, а именно после двоеточия пишем:

hours(0), minutes(0).

По сути, это конструктор для простых типов данных, который введен в С++ для единообразия описания классов и простых типов. Тело же конструктора останется пустым, поскольку делать больше ничего не нужно. Можно было описать конструктор так:

Time(){hours = minutes = 0;}

но конструктор, приведенный выше, считается предпочтительнее.



Так можно инициализировать переменные только в конструкторе и описании переменной.

Однако, если бы все ограничивалось конструктором по умолчанию, нам пришлось бы строить метод для присваивания переменным значения. У нас есть необходимость создавать объект Time с заданием hours, а также задавать сразу оба параметра: hours, minutes. Проблема здесь заключается в том, что конструктор класса должен однозначно определяться компилятором, поэтому разработчиками языка принято простое решение — имя конструктора должно совпадать с именем класса, однако для идентификации конструктора, как и любой другой функции, компилятор использует не только имя, но также тип и количество параметров. Таким образом, нет никаких проблем в создании нескольких функций с одинаковыми именами, но различающихся типом и количеством параметров.

Опишем еще один конструктор с двумя параметрами, который позволит за- дать значение часов и минут при создании объекта.

Time(int h, int m = 0): hours(h), minutes(m) { }

Здесь мы немного сэкономили и вместо того, чтобы сперва описать конст- руктор с одним параметром, а только потом с двумя, совместили все в одном

описании. Просто воспользовались возможностью задания значения по умол- чанию для второго параметра. Таким образом, этот конструктор будет рабо- тать как с одним параметром, так и с двумя. Теперь можно описывать пере- менные так:

Time t1(12), t2(17,15);

В первой переменной установлено время 12:00, во второй — 17:15.

И наконец, опишем конструктор, который принимает в качестве параметра переменную типа своего же класса. Это так называемый *конструктор копи- рования*, и это один из тех случаев, когда значение должно приниматься только по ссылке. Действительно, для организации обращения к этому конст- руктору при передаче экземпляра класса по значению компилятор должен построить копию этого экземпляра в стеке, но это можно сделать только при помощи конструктора копирования, который еще не создан. При передаче же параметра по ссылке фактически произойдет передача указателя на объект.

Time(Time& t): hours(t.hours), minutes(t.minutes) { }

Здесь видно, как мы получаем доступ к переменным экземпляра класса, а именно t.hours, t.minutes точно так же, как и к элементам структуры.

Мы рассмотрели описание трех конструкторов класса Time. *Конструкто- ры* — это специальные методы класса, они отличаются от прочих двумя ос- новными признаками: во-первых, имя конструктора совпадает с именем клас- са, а во-вторых, — они не имеют возвращаемого значения. Как же компиля- тор разберется, какой конструктор должен работать? А очень просто, по количеству и типу параметров. Если переменная объявляется без параметров, то работает конструктор по умолчанию, если используются один или два па- раметра — конструктор со значением по умолчанию, если же в качестве па- раметра выступает экземпляр этого же класса или же при описании использу- ется явное присваивание, например:

Time k = tw;

копирующий конструктор. Все это называется *перегрузкой методов*. Далее два метода, возвращающие часы и минуты в виде целого числа:

int gethours() { return hours;}

int getminutes() { return minutes; }

Методы выглядят очень просто и состоят из одного оператора return, кото- рый возвращает соответственно часы и минуты. Это издержки объектно- ориентированного подхода, который запрещает прямой доступ к переменным класса. Обычно столь простые методы объявляют как inline-функции, что служит рекомендацией компилятору вместо построения функции встроить соответствующий код.

Эти методы можно было описать так:

inline int gethours() { return hours;} inline int getminutes() { return minutes; }

В этом случае компилятор сам определяет оптимальный вариант и либо встраивает код, либо игнорирует наше пожелание и строит метод обычным способом.

Метод set() устанавливает значение переменных, причем предусмотрены значения по умолчанию, что позволит обращаться к методу как без парамет- ров, так с одним или двумя параметрами.

void set(int h = 0, int m = 0)

{ hours = h; minutes = m; }

Следующий метод добавляет минуты к переменной minutes. Нужно учесть случай, когда суммарное количество минут превысит 60. В этом случае мы добавляем часы, а минуты получаются как остаток от деления минут на 60.

void addmin(int m)

{ minutes += m;

hours += minutes/60; minutes %= 60;

}

Часы прибавляем в методе addhours(). void addhours(int h){ hours += h; }

Для отображения результата создадим метод show(). Здесь мы используем манипулятор потока вывода setw(2), который обеспечит вывод в поле разме- ром в два символа. Однако для его использования необходимо подключить заголовочный файл iomanip.h, где описаны все потоковые манипуляторы. (Про манипуляторы потока ввода поговорим подробнее в *главе 3*, когда бу- дем рассматривать потоковый ввод/вывод.)

void show()

{ cout<<setw(2)<<hours<<':'<<setw(2)<<minutes<<endl;}

И, наконец, перегрузим оператор + для класса Time. Как видно из примера, *перегрузка оператора* отличается от описания метода лишь тем, что вместо имени метода мы пишем operator +. Это называется перегрузкой оператора, потому что можно переопределить лишь существующий оператор и нельзя ввести новый. Оператор должен вернуть экземпляр класса Time, поэтому мы описываем переменную типа Time, определяем ее поля и возвращаем резуль- тат оператором return.

Time operator + (Time& t)

{ Time tmp;

tmp.minutes = minutes + t.minutes;

tmp.hours = hours + t.hours + tmp.minutes/60; tmp.minutes %= 60;

return tmp;

}

Вообще-то обычно такие операторы имеют возвращаемое значение по ссыл- ке, но мы рассмотрим этот вариант чуть позже.

Формализм языка предполагает, что при перегрузке оператора, по умолча- нию, в качестве первого операнда используется текущий экземпляр класса. Теперь мы вправе использовать выражения типа tw + tf.



На самом деле, при построении любого метода компилятор в качестве первого параметра автоматически ставит ссылку на текущий объект.

Обычно, при разработке больших классов, описание класса и реализацию ме- тодов строят в разных файлах, поэтому построим, для примера, этот же класс, используя для методов отдельный файл реализации. В листинге 2.3 представ- лен заголовочный файл Time.h, в листинге 2.4 — файл реализации Time.cpp.

**Листинг 2.3. Файл Time.h**

#include <iostream.h> #include <iomanip.h> class Time

{

private: int hours;

int minutes;

public:// Здесь только объявления методов Time();

Time(int h, int m = 0); inline int gethours(); inline int getminutes(); void addmin(int m); void addhours(int h);

void set(int h = 0, int m = 0); void show();

Time operator + (Time t);

};

**Листинг 2.4. Файл Time.cpp**

#include "Time.h"

Time::Time(): hours(0), minutes(0) { }

Time::Time(int h, int m): hours(h), minutes(m) { } inline int Time::gethours() { return hours;} inline int Time::getminutes() { return minutes; } void Time::addmin(int m)

{

minutes += m;

hours += minutes/60; minutes %= 60;

}

void Time::addhours(int h) { hours += h;} void Time::set(int h, int m)

{

hours = h; minutes = m;

}

void Time::show()

{

cout << setw(2) << hours << ':' << setw(2) << minutes << endl;

}

Time Time::operator + (Time t)

{

Time tmp;

tmp.minutes = minutes + t.minutes;

tmp.hours = hours + t.hours + tmp.minutes/60; tmp.minutes %= 60;

return tmp;

}

Особенность описания методов в файле реализации заключается в необходи- мости указывать явно *область видимости*. Перед именем каждого метода нужно указывать ее в виде Time::. В этом случае все переменные класса дос- тупны внутри метода и без явного указания.

В головной функции main(), которая работает без изменений для обоих спо- собов реализации класса, мы определяем три экземпляра класса и проверяем работу всех созданных методов. Отметим только, что мы использовали мани- пулятор шаблона заполнения setfill('0') в головной функции, поскольку он действует на весь последующий вывод, в отличие от манипулятора setw(2), который действует только на одно выводимое значение: cout << setfill('0').

**Класс *String***

В качестве более серьезного примера рассмотрим класс String, он хорошо демонстрирует синтаксис и основные черты классов С++. Хотя существует стандартный класс string, но создание своего класса позволит обсудить мно- гие проблемы ООП. Кроме того, здесь мы решим проблему использования русского алфавита, что в консольном приложении Visual C++ 6.0 является проблемой.

Под строковой переменной в классическом С понимается символьный мас- сив, заканчивающийся символом '\0'. Мы же создадим класс, в котором бу- дем хранить лишь указатель типа char\* и отдельно, в виде целого числа, бу- дем хранить размер символьного массива без завершающего символа '\0', а память под строку будем выделять при создании переменной. Это закрытые переменные класса.

Описываем класс, снабжая его необходимыми комментариями, для простоты совместив описание методов с их реализацией. Так нам будет удобнее в дальнейшем, когда мы будем подключать описание этого класса к другим проектам.

#include <iostream.h> #include <windows.h> class String

{

private:

char \*s; // Указатель на строку int n; // Размер строки

Добавим статическую функцию для подсчета размера текстовой строки, что- бы не пользоваться стандартной функцией из библиотеки string.h. Функция будет использоваться лишь методами класса, поэтому она лежит в закрытой области.

// Служебная функция - длина строки static int strlen(char \*str)

{ for (int i = 0; \*str++; i++); return i;

}



Служебное слово static в описании функции класса означает, что она принад- лежит всему классу, а не отдельному его экземпляру. В этом случае обращение к ней в описании методов класса строится как к обычной функции, а вне класса, с указанием имени класса, как области видимости.

Определим несколько конструкторов класса в открытой области класса. Кон- структор по умолчанию просто обнуляет указатель s и размер массива n. Также определим конструктор, принимающий текстовую строку и форми- рующий экземпляр класса String. Для этого мы выделяем необходимую па- мять и копируем туда символьный массив. И, наконец, копирующий конст- руктор, принимающий ссылку на объект String. Он выделяет память и созда- ет копию объекта. Передача параметра здесь осуществляется по ссылке, причина этого обсуждалась ранее.

public:

// Конструктор по умолчанию String(): s(0), n(0) {}

// Конструктор принимает строку текста String(char \*str)

{ n = strlen(str); s = new char[n];

for (int i = 0; i < n; i++) s[i] = str[i];

}

// Копирующий конструктор принимает экземпляр String String(String& T)

{n = T.n;

s = new char[n];

for (int i = 0; i < n; i++) s[i] = T.s[i];

}

Здесь имеется необходимость описать *деструктор* — специальный метод класса, который автоматически вызывается при уничтожении объекта. Имя деструктора совпадает с именем класса со знаком ~ (тильда) в начале. Задача деструктора — освободить ресурсы, занятые объектом, перед его уничтоже- нием. Действительно, если просто удалить экземпляр класса String, то па- мять, выделенная для строки текста, останется занятой и не сможет быть ос- вобождена, поскольку указатель на эту область памяти уничтожен. Таким образом, мы можем "замусорить" память удаленными объектами, поэтому, если экземпляр класса получает какие-либо ресурсы, необходим деструктор, освобождающий эти ресурсы. Поскольку при создании экземпляра класса выделяется память, нам необходим деструктор, который освободит выделен- ную память.

// Деструктор освобождает память, занятую текстом virtual ~String() { if(s) delete[] s; }

Мы использовали условный оператор if, поскольку конструктор по умолча- нию память не выделяет, лишь обнуляет указатель, и освобождать в этом случае нечего. Можно было бы опустить оператор if, поскольку освобожде- ние памяти по нулевому указателю безопасно для системы.

Модификатор virtual создается автоматически и его можно убрать, но он будет важен, если мы хотим использовать класс String как родительский.

Для того чтобы иметь возможность выполнять операции типа str1 = str2;, необходимо перегрузить операцию присваивания. Без перегрузки оператора такое присваивание приведет к присваиванию указателей, что означает ссыл- ку двух указателей на один строковый массив, и создаст ошибочную ситуа- цию при попытке деструктора освободить несуществующую память. Дейст- вительно, при удалении первого экземпляра класса, например str1, будет ос- вобождена память, на которую показывает указатель str1.s, и при вызове деструктора второго экземпляра класса, например str2, будет осуществлена попытка еще раз освободить эту же память, что неизбежно приведет к сис- темной ошибке.

Очевидно, появляется необходимость ввести специальный указатель this, который указывает на текущий экземпляр класса. Тогда \*this — это сам те- кущий экземпляр класса.

// Перегрузка оператора присваивания String& operator = (String& st)

{// Проверка на самоприсваивание if (&st == this) return \*this;

if (s) delete[] s; // Удаление старого содержимого n = st.n;

s = new char[n]; // Выделение памяти

for (int i = 0; i < n; i++) s[i] = st.s[i]; // Копирование return \*this;

}

Что важно в перегрузке оператора присваивания, так это возвращаемое зна- чение в виде экземпляра класса String, причем по ссылке. Это позволяет ис- пользовать многократное присваивание типа s1 = s2 = s3, а возвращение значения по ссылке является общепринятой практикой и позволяет избегать передачи через стек больших блоков данных.

Приступим к описанию необходимых методов. Самый простой метод воз- вращает размер текста в экземпляре String.

int length() { return n; }

Извлечение символа с индексом k.

char at(int k) { return (k >= 0 && k < n)? s[k] : '\0'); }

Установить символ.

void setchar(char ch, int k) {if (k >= 0 && k < n) s[k] = ch;}

Для двух предыдущих методов вполне логично проверить, не выходит ли ин-

декс за границы строки *k* [0, *n*)? И если для функции at() вполне логично

вывести нулевой байт в случае ошибки, то функция setchar() просто ничего не делает.

Удаление фрагмента строки со склеиванием.

void erase(int start, int end)

{ int i, j, k = n + start - end -1; char \*p = new char[k];

for (i = 0; i < start; i++) p[i] = s[i];

for (i--,j = end; i < k; i++, j++) p[i] = s[j]; delete[] s;

s = p; n = k;

}

Мы вычисляем новую длину строки и выделяем под нее память, затем копи- руем туда начало строки и дописываем далее остаток строки. После этого память под исходной строкой удаляется, указатель новой строки присваива- ется переменной s экземпляра класса, а размер строки — переменной n.

Вставка строки с k-й позиции объекта String (текст раздвигается).

void insert(int k, char \*str)

{ int i, j, len = strlen(str); char \*p = new char[n + len];

for (i = 0; i < k; i++)p[i] = s[i]; // Начало строки

for (j = k; j < k + len; j++) p[j] = \*str++; // Вставка n += len;

for (; j < n; i++,j++) p[j] = s[i]; // Конец строки delete[] s; // Освобождаем память, занятую текстом s = p;

}

Здесь поступаем аналогично предыдущему методу, только строку str добав- ляем к тексту с k-й позиции.

Добавление строки к концу объекта String.

void append(char \*str)

{ int len = strlen(str); int i, k = n + len; char \*p = new char[k];

for (i = 0; i < n; i++) p[i] = s[i]; for (; i < k; i++) p[i]= \*str++;

delete[] s; s = p;

n = k;

}

Метод отличается от метода insert() только тем, что строка добавляется в конец текста.

Чтобы иметь возможность использовать класс String с контейнерами стан- дартной библиотеки шаблонов STL, нужно перегрузить операторы сравнения

==, !=, <, >. Понятие "одна строка больше другой" здесь рассматривается в лексикографическом смысле, т. е. если код символа одной строки больше, чем код символа другой строки, то строка считается большей. Если размеры строк не совпадают, то любой символ больше отсутствующего символа. На- помним, что при перегрузке оператора в качестве первого операнда по умол- чанию используется текущий экземпляр класса \*this.

// Сравнение объектов String == bool operator == (String& s2)

{ if (n != s2.n) return false; // Размеры строк должны совпадать for (int i = 0; i < n; i++) if (s[i] != s2.s[i]) return false;

return true;

}

// Сравнение объектов String != bool operator != (String& s2)

{

return !(\*this == s2);

}

// Сравнение объектов String < bool operator < (String& s2)

{

int k = (n < s2.n)? n : s2.n; for (int i = 0; i < k; i++)

{ if (s[i] == s2.s[i]) continue;

return (s[i] < s2.s[i])? true : false;

}

return (n < k)? true : false;

}

// Сравнение объектов String >

bool operator > (String& s2)

{

return !(\*this < s2 || \*this == s2);

}

Перегрузка оператора + для "склеивания" объектов String. После описания оператора возможна операция s = s1 + s2;, где все переменные типа String. Здесь принципиально важно создать экземпляр класса String оператором

String \*temp = new String;

поскольку только в этом случае объект не будет уничтожен при выходе из метода и может быть использован в качестве возвращаемого значения, кото- рое принимается перегруженным оператором присваивания.

String& operator + (String& str)

{ int i,j, k = n + str.n; char \*p = new char[k]; String \*temp = new String;

for (i = 0; i < n; i++) p[i] = s[i];

for (j = 0; i < k; i++,j++) p[i] = str.s[j]; temp->s = p;

temp->n = k; return \*temp;

}

Некоторыми особенностями обладает оператор преобразования типа для эк- земпляра класса. Поскольку тип преобразования указывается явно, то он и является возвращаемым значением и не описывается как тип возвращаемого значения.

Формат оператора преобразования типа:

operator тип () {

. . .

return возвращаемое значение;

}

В реализации оператора нам необходимо выделить память под строку, ско- пировать туда символьный массив из экземпляра класса String, не забывая добавить признак конца строки '\0', и вернуть указатель на строку.

// Преобразование типа: String в строку char\*. operator char\* ()

{ char \*p = new char[n+1];

for (int i = 0; i < n; i++) p[i] = s[i]; p[i] = '\0';

return p;

} // Сейчас возможно выражение: char \*str=s; где s типа String.

Мы перегрузим оператор вывода на консоль для того, чтобы иметь возмож- ность выводить переменную s типа String простой операцией cout << s. По-

путно решим проблему с выводом кириллического текста. Проблема здесь заключается в том, что в консольном приложении Visual C++ 6.0 принята кодировка ANSI, нам же при выводе необходимо преобразовать ее в стандартный ASCII-код. Это можно сделать при помощи функции CharToOem(ANSI,ASCII), прототип которой помещен в файл включений windows.h. К сожалению, функция не имеет возвращаемого значения, поэтому мы описываем два промежуточных массива типа char, проводим преобразо- вание и выводим преобразованный текст на консоль. Нужно также добавить и файл включений iostream.h для операторов потокового ввода/вывода.

Здесь нужно решить две проблемы:

* Первая проблема заключается в том, что оператор вывода не может быть членом класса, поскольку по имеющемуся соглашению в выражении:

операнд1 ОПЕРАЦИЯ операнд2

первый операнд должен быть экземпляром класса, но мы имеем выраже- ние cout<< s, и здесь первым операндом является потоковая переменная cout. Решение проблемы может быть лишь в перегрузке оператора, не яв- ляющегося членом класса, но в таком случае мы не можем описывать его в классе и не имеем доступа к закрытым переменным класса. Можно, ко- нечно, получить доступ к значению переменных при помощи специаль- ных методов, но есть более элегантное решение этой проблемы. Доста- точно объявить оператор *дружественным* (служебное слово friend), и тогда мы можем напрямую получить доступ к закрытым членам класса и опи- сать оператор в классе.



Есть мнение, что введение дружественных операторов и функций разрушает концепцию объектно-ориентированного программирования, и их использование должно быть ограничено.

* Вторая же проблема заключается в том, что использовать в качестве воз- вращаемого значения? Можно вообще ничего не возвращать, но в этом случае невозможно будет использовать выражения типа:

cout << s1 << s2 << endl;

Действительно, для того чтобы работал оператор << s2, необходимо слева иметь потоковую переменную cout. То есть выражение cout << s1 должно вновь вернуть поток вывода, а потоки принято передавать по ссылке. (По значению поток передать вообще нельзя.)

В качестве параметров оператора будет поток вывода и экземпляр класса

String. Оба параметра будем передавать по ссылке.

friend ostream& operator << (ostream& out, String& str)

{

char \*st = str; // Используем неявное преобразование типа char \*BufRus = new char[str.n+1];

CharToOem(st,BufRus); // Перекодировка ANSI->ASCII out << BufRus;

delete[] BufRus; // Освобождение памяти return out;

}

Перегрузка оператора ввода с консоли будет реализована симметрично опе- ратору вывода, только вместо функции CharToOem(ANSI,ASCII) будет исполь- зована функция OemToChar(ASCII,ANSI), обеспечивающая обратное преобразо- вание. Нам остается лишь выделить необходимую область памяти для текста и скопировать туда символьный массив. Для этих операторов возвращаемым значением является поток, что обеспечивает возможность агрегатного ис- пользования, например: cin >> s1 >> s2;

friend istream& operator >> (istream& in, String& str)

{

char ch, st[256], BufRus[256]; in.getline(st,256); // Вводим строку int i = strlen(st);

OemToChar(st,BufRus); // Перекодировка ASCII->ANSI char \*p = new char[i];

for (int j = 0; j < i; j++) p[j] = BufRus[j]; str.s = p;

str.n = i; return in;

}

};

Конечно, можно добавить еще методы, перегрузить еще несколько операто- ров, но это уже непринципиально. На примере разработки этого класса мы рассмотрели описание переменных, статических функций, создание методов и перегрузку операторов. Можно найти более эффективные решения, но здесь на первом месте стояла ясность изложения. Также совсем не рассмат- ривались вопросы обработки ошибочных ситуаций, что вообще-то нелишне. Все эти вопросы мы оставляем для дальнейшего рассмотрения. Далее, в лис- тинге 2.5, приведен листинг тестирующей головной функции.

|  |
| --- |
| **Листинг 2.5. Тестирующая головная функция** |

#include "String.h" void main()

{

String str, t("String main"), text(t); // Три конструктора

// Безымянный объект String преобразуется в строку char \*p = String(" add");

cout << t << endl;

text.append(p); // Добавить строку cout << text << endl;

String s("1");

str = text + s; // Склеивание строк cout << str << endl;

// Заменим пробелы подчеркиванием

for (int i = 0;i < text.length(); i++)

if (text.at(i) == ' ') text.setchar('\_',i); cout << text << endl;

text.erase(7,11); // Удалить слово cout << text << endl;

cout << String("Русский текст: ");

cin >> t; // Ввести русский текст cout << t << endl;

}

## Пространство имен

При разработке сложных программных комплексов группами программистов часто возникает проблема совпадения имен переменных в различных частях программы, иногда и в небольшой программе удобнее использовать некие стандартные имена объектов. Для разрешения коллизий, возникающих в по- добных ситуациях, введено понятие *пространства имен*. По умолчанию про- странством имени переменной является блок, ограниченный фигурными скобками, в котором эта переменная объявлена. Для переменной, объявлен- ной на глобальном уровне, пространством имен будет весь файл. Например:

const int N = 10; // Область действия – до конца файла

{ int number; // Область действия – до закрывающейся скобки

...

}

Можно, однако, явно объявить собственное пространство имен директивой:

namespace имя

{

...

}

Например:

namespace user

{ int k, r;

...

}

Доступ к переменным из именованной области осуществляется следующими способами:

1. Можно использовать директиву using, которая открывает доступ к пере- менным именованного блока, как к собственным именам, например:

using namespace user; int m = k/2;

1. Можно сделать доступной отдельную переменную из другой области ви- димости, например:

using user:: k;

1. Можно для каждой переменной указывать ее область видимости директи- вой имя::, например:

int m = user::k/2;

Специальное имя "::" означает глобальный уровень. Например, пусть у нас имеется переменная, объявленная как на глобальном уровне, так и в отдель- ном блоке:

int k = 1;

{ int k = 2;

int m = k; // m = 2; локальная переменная int q = ::k; // q = 1; глобальная переменная

...

}

Подробнее с этими вопросами мы будем знакомиться по мере необходимо- сти, пока же нас интересует *стандартное пространство имен* std, где раз- мещена стандартная библиотека языка С++. Для того чтобы иметь доступ к области имен стандартной библиотеки, нам нужно написать директиву:

using namespace std;

Теперь, если нам необходим доступ к области стандартных имен, мы все свои программы будем начинать с этой директивы. Но здесь имеется одна пробле- ма, дело в том, что для совместимости с классическим языком С разработчи- ки С++ сохранили старые С-библиотеки, но эти старые функции могут кон- фликтовать в области имен std. Для разрешения таких коллизий старые биб-

лиотеки были переработаны и получили новые имена, из которых исключено расширение "h", а в начале имени добавлен префикс "с" — си, так библиотека string.h стала называться cstring.

Для классов объявлять область имен не имеет смысла, поскольку, по умолча- нию, имя класса — это и есть имя этой области. Часто возникает ситуация, когда одна область имен включена внутрь другой, в этом случае полное имя указывается последовательным перечислением всей иерархии имен, напри- мер: std::ios::eof();.

## Наследование

Рассмотрим еще одно важное свойство классов — это способность *наследо- вания*. Класс может наследовать переменные и методы другого класса и об- ращаться с ними как с собственными переменными и методами. Рассмотрим, например, такие схемы:

|  |  |
| --- | --- |
| class A | class A |
| { | { |
| . . . | . . . |
| }; | }; |
| class B : public A | class B |
| { | { |
| . . . | . . . |
| }; | }; |
| class C : public B | class C : public A, public B |
| { | { |
| . . . | . . . |
| }; | }; |

Смысл этих схем достаточно прозрачен: в первом случае класс B является на- следником класса A, а класс C наследует от B; во втором случае классы A и B независимы, а класс C является наследником сразу двух классов *(множест- венное наследование)*. И в том, и в другом случае результирующие классы C будут идентичны.

### Наследование на примере классов: Работник -> Менеджер -> Ученый

Описание всех трех классов поместим в одном файле. Вначале опишем класс Employee, где введем переменную name типа String для того, чтобы можно было бы использовать русский текст. Действительно, после создания класса

String мы можем использовать переменные этого класса наравне с перемен- ными простых типов, необходимо только добавить файл включений String.h. (В этом случае подключать файл iostream.h не нужно, поскольку он уже при- сутствует в String.h.) Разумеется, сам файл с описанием класса должен быть скопирован в папку проекта, поскольку директива #include "String.h" под- разумевает поиск этого файла в текущей папке. Вторая переменная целого типа — номер работника, а поскольку отрицательный номер выглядел бы не совсем корректно, есть смысл описать его как беззнаковое целое, т. е. unsigned int. Обратите внимание на то, что вместо того, чтобы объявить пе- ременные как закрытые private, мы объявляем их защищенными protected. Это почти то же самое, с единственным отличием — *защищенные члены класса* доступны при наследовании, а поскольку мы собираемся сделать класс Employee базовым, то это имеет смысл.

Далее опишем два конструктора: конструктор по умолчанию и конструктор с двумя параметрами. Следует обратить внимание на то, что инициализация переменной типа String name(nam) осуществляется при помощи конструктора копирования класса String. Деструктор здесь ничего делать не будет, а для переменной name работает ее собственный деструктор.

Мы опишем лишь два метода для ввода и вывода информации: get() и put() соответственно. И столкнемся с проблемой, связанной с особенностями пото- кового ввода, а именно при вводе числового значения cin >> number; фор- матный оператор >> вводит число до первого нечислового символа, оставляя его в потоке (в нашем случае, символ перевода строки '\n'). Вследствие это- го, следующий оператор ввода получит пустую строку. Чтобы избежать по- добного недоразумения, можно использовать метод ignore() класса cin без параметров, который обеспечит пропуск одного символа. Для вывода на кон- соль комментариев на русском языке воспользуемся неименованными экзем- плярами класса String, например String("Фамилия: ");. В этом случае созда- ется объект класса String, выводится на консоль перегруженным оператором

<< и тут же уничтожается. Далее, в листинге 2.6, приведен текст класса

Employee.

|  |
| --- |
| **Листинг 2.6. Класс Employee** |

#include "String.h"

class Employee // Класс Работник

{

protected:

String name; unsigned int number;

public:

Employee() {}

Employee(String nam, unsigned int num):name(nam),number(num){} virtual ~Employee() {}

void get()

{

cout << String("Фамилия: "); cin >> name;

cout << String("Номер: "); cin >> number;

cin.ignore(); // Убираем лишний символ '\n'

}

void put()

{

cout << String("Фамилия: ") << name << endl; cout << String("Номер: ") << number << endl;

}

};

Теперь будем строить класс Manager, как наследник класса Employee.

*Наследование* объявляется в заголовке класса:

class Manager : public Employee

Служебное слово public означает, что члены класса Employee наследуются с теми же правами доступа.

Здесь мы опишем одну переменную title типа String для хранения инфор- мации о должности менеджера. Опишем также два конструктора, причем следует обратить внимание на то, как мы инициализировали переменные на- следуемого класса: мы просто воспользовались конструктором базового класса Employee(nam,num), а переменная title инициализируется конструкто- ром класса String.

В методе get() для ввода переменных наследуемого класса Employee мы вос- пользуемся уже готовым методом этого класса, но нам придется задать об- ласть его видимости, т. е. написать так: Employee::get();. Оставшуюся пере- менную введем как обычно.

В методе put() поступим аналогично. В листинге 2.7 приведен текст класса

Manager.

|  |
| --- |
| **Листинг 2.7. Класс Manager** |

class Manager : public Employee // Класс Менеджер

{

protected: String title;

public:

Manager() { } virtual ~Manager() {}

Manager(String nam, unsigned int num, String tit): Employee(nam,num), title(tit) {}

void get()

{

Employee::get();

cout << String("Должность: "); cin >> title;

}

void put()

{

Employee::put();

cout << String("Должность: ") << title << endl;

}

};

Далее строим класс Scientist, как наследник класса Manager. Здесь мы объяв- ляем единственную переменную целого типа pubs (количество публикаций) с правами доступа private, поскольку далее цепочку наследования продол- жать не будем. Создадим два конструктора и два метода get() и put() так же, как мы это делали для предыдущего класса. Текст класса приведен в листин- ге 2.8.

|  |
| --- |
| **Листинг 2.8. Класс Scientist** |

class Scientist : public Manager // Класс Ученый

{

private: int pubs; public: Scientist() {}

Scientist(String nam, unsigned int num, String tit, int pub): Manager(nam, num, tit), pubs(pub) { }

void get()

{

Manager::get();

cout << String("Количество публикаций: "); cin >> pubs;

cin.ignore(); // Убираем лишний символ '\n'

}

void put()

{

Manager::put();

cout << String("Количество публикаций: ")<< pubs << endl;

}

};

Несмотря на то, что класс Manager является родительским классом для класса Scientist, мы можем создавать переменные этого класса и пользоваться его методами наравне с переменными и методами порожденного класса. Это ил- люстрирует простая головная функция (листинг 2.9), в которой создается по одному объекту классов Manager и Scientist, а также демонстрируется работа созданных нами методов.

|  |
| --- |
| **Листинг 2.9. Головная функция** |

#include "Employee.h" void main()

{

Manager mn; mn.get();

mn.put(); Scientist s; s.get();

s.put();

}

### Использование классов,

**как пользовательских типов данных**

Решение предыдущей задачи могло быть построено и без использования на- следования, вместо этого в конечном классе Scientist мы могли бы исполь- зовать в качестве переменных экземпляры созданных нами классов. Услож- ним немного задачу и введем еще один класс Student — для хранения информации об уровне образования. Однако если мы хотим создать конст- руктор класса Scientist, тогда, для инициализации переменных исходных классов, необходимо создать копирующие конструкторы. Новые классы при- ведены в листинге 2.10.

**Листинг 2.10. Новые классы Employee, Student**

#include "String.h" class Employee

{

private:

String name; unsigned int number;

public:

Employee() {}

// Копирующий конструктор

Employee(Employee& e): name(e.name), number(e.number) {} void get()

{

cout << String("Фамилия: "); cin >> name; cout << String("Номер: "); cin >> number; cin.ignore();

}

void put()

{

cout << String("Фамилия: ") << name << endl; cout << String("Номер: ") << number << endl;

}

};

/////////////////////////////////////////////////////////////////// class Student

{

private:

String school;

unsigned int degree; // 11-средняя школа, 12-колледж и т. д....

public:

Student() { }

// Копирующий конструктор

Student(Student& st) : school(st.school), degree(st.degree){} void get()

{

cout << String("Название учебного заведения: "); cin >> school;

cout << String("Уровень образования (11 - ср.школа,12 -\ колледж,

...: ");

cin >> degree; cin.ignore();

}

void put()

{

cout<<String("Название учебного заведения: ")<<school<<endl; cout<<String("Уровень образования: ")<<degree<<endl;

}

};

Сейчас в качестве переменных класса Scientist мы используем экземпляры классов Employee, Student, String, а также переменную простого типа int. При создании конструктора с 4 параметрами для инициализации экземпляров

классов без их копирующих конструкторов никак не обойтись, поэтому мы их и добавили в исходные классы. В классе String копирующий конструктор у нас уже имелся, а инициализация простого типа осуществляется системны- ми средствами. В методах get() и put() мы можем использовать аналогичные методы классов Employee и Student, вызвав их через соответствующие экзем- пляры класса, например, empl.get(); std.get(). В листинге 2.11 представлен текст класса Scientist без использования наследования, а головная функция приведена в листинге 2.12.

**Листинг 2.11. Класс Scientist без использования наследования**

class Scientist

{

private: Employee empl;

Student std; String title; int pubs;

public: Scientist() {}

Scientist(Employee emp, Student st, String tit, int pub): empl(emp), std(st), title(tit), pubs(pub) { }

void get()

{

empl.get();

std.get();

cout << String("Должность: "); cin >> title;

cout << String("Количество публикаций: "); cin >> pubs; cin.ignore();

}

void put()

{

empl.put();

std.put();

cout << String("Должность: ") << title << endl;

cout << String("Количество публикаций: ")<< pubs << endl;

}

};

**Листинг 2.12. Головная функция**

#include "Employee.h" void main()

{

Scientist s; s.get();

s.put();

}

Вопрос, каким образом строить классы — использовать наследование или же строить сложные типы данных, решается индивидуально в каждом конкрет- ном случае, и дать какие-либо общие рекомендации не представляется воз- можным. Часто применяют одновременно оба подхода, как мы и поступили изначально, используя в качестве типа данных переменную типа String.

## Виртуальные методы и классы

Метод класса может быть объявлен как *виртуальный* спецификатором virtual. (Виртуальный — кажущийся, т. е. видимый, но реально не существующий.) Чтобы разобраться, зачем это нужно, рассмотрим в качестве примера класс Base с единственным методом show(), который выводит на консоль сообще- ние "Base". Рассмотрим два класса, полученных в результате наследования: Dev1, Dev2, также имеющих по одному методу show(). Все эти классы приве- дены в листинге 2.13.

**Листинг 2.13. Классы Base, Dev1, Dev2**

#include <iostream.h> class Base

{ public: void show() { cout << "Base\n"; }

};

class Dev1: public Base

{ public: void show() { cout << "Dev1\n"; }

};

class Dev2: public Base

{ public: void show() { cout << "Dev2\n"; }

};

В головной функции (листинг 2.14) создадим две переменных типа Dev1 и Dev2, а также указатель типа Base. Теперь, присвоив этому указателю адреса переменных dv1 и dv2, выполним для него метод show().



Указателю базового класса разрешается ссылаться на объекты порожденных классов.

**Листинг 2.14. Головная функция**

#include "Base.h" void main()

{ Dev1 dv1; Dev2 dv2; Base \*p; p = &dv1; p->show(); p = &dv2; p->show();

}

Вывод:

Base Base

Наверное, нет ничего удивительного в том, что в результате выполнения про- граммы работал метод базового класса, поскольку указатель p принадлежит этому классу. Однако если мы добавим спецификатор virtual перед методом базового класса

virtual void show() { cout << "Base\n"; }

ситуация кардинально изменится.



В результате и методы порожденных классов также будут виртуальными, по- этому служебное слово virtual в их реализации может быть опущено, но его часто пишут, чтобы подчеркнуть, что мы имеем дело с виртуальными мето- дами.

На консоль будет выведено:

Dev1 Dev2

Дело в том, что при объявлении метода виртуальным выбор реализации ме- тода в выражении p->show() переносится с этапа компиляции на этап выпол- нения программного кода *(позднее связывание)*. Реализация этой технологии осуществляется путем построения компилятором таблицы виртуальных функций, откуда на этапе выполнения и выбирается соответствующая реали- зация. В нашем случае, поскольку указатель p последовательно перебирает экземпляры классов Dev1 и Dev2, то и работают методы этих классов. Сейчас

нам нет необходимости в реализации метода show() для класса Base, поэтому его можно объявить *чисто виртуальным*. Это достигается объявлением:

virtual void show() = 0;

где операция = 0 имеет чисто символическое значение и служит указанием компилятору считать метод чисто виртуальным. Классы, содержащие чисто виртуальные методы, называются *виртуальными* и используются только для наследования, как родительские классы. В порожденных классах чисто вир- туальные методы должны быть переопределены, иначе и порожденный класс останется виртуальным. Особенностью виртуальных классов является невоз- можность создавать объекты этих классов, но указатель виртуального класса объявить можно.

В классах, имеющих *виртуальные функции*, и деструкторы должны быть вир- туальными. Чтобы проиллюстрировать это утверждение, опишем деструкто- ры ранее рассмотренных классов:

class Base

{public: ~Base() {cout << "Destructor Base\n"; }

};

class Dev1: public Base

{public: ~Dev1() {cout << "Destructor Dev1\n"; }

};

В головной функции (листинг 2.15) опишем указатель типа Base и создадим объект Dev1. Теперь попытаемся освободить выделенную объекту Dev1 па- мять. Как видим, работает только деструктор базового класса, и если конст- руктор класса Dev1 выделил память, то ссылки на нее будут утеряны.

|  |
| --- |
| **Листинг 2.15. Головная функция** |

void main()

{

Base \*p;

p = new Dev1; delete p;

}

Вывод:

Destructor Base

Теперь объявим деструктор ~Base() виртуальным. Как можно видеть, вначале работает *деструктор порожденного класса*, затем — базового.

class Base

{public: virtual ~Base() {cout << "Destructor Base\n"; }

};

. . . . . . . .

Вывод:

Destructor Dev1 Destructor Base

Из этого примера становится понятно, зачем среда программирования с усер- дием, достойным лучшего применения, автоматически строит деструктор виртуальным. Если наследников этого класса не будет, то вреда от этого не- много, а если будут порожденные классы, то это позволит избежать проблем с потерей памяти. Хотя, если наследования не предполагается, деструктор лучше не делать виртуальным.

### Классы, объявленные как виртуальные

Иногда, для разрешения коллизий наследования, необходимо объявлять клас- сы виртуальными. Рассмотрим для примера помещенную на рис. 2.3 схему наследования, реализация которой приведена далее в листинге 2.16.



**Рис. 2.3.** Схема наследования

|  |
| --- |
| **Листинг 2.16. Задача с двойным наследованием** |

class Parent

{

protected: int data; public: Parent(): data(1){}

};

class Child1: public Parent {}; class Child2: public Parent {};

class Grandchild: public Child1, public Child2

{

public:

Grandchild() {}

int getdata() { return data;}

};

#include "Parent.h" #include <iostream.h> void main()

{

Grandchild g;

cout << g.getdata()<< endl;

}

Если реализовать такую схему, компилятор выдаст ошибку при создании эк- земпляра класса Grandchild. Причина этого очевидна, поскольку каждый из экземпляров классов Child1 и Child2 содержит *родительский класс* Parent. Компилятор, в этом случае не сможет разрешить проблему — из какого клас- са выбрать информацию о классе Parent. Однако если объявить, что насле- дуемые классы содержат класс Parent как виртуальный:

class Child1: virtual public Parent {}; class Child2: virtual public Parent {};

проблема разрешается, и класс Grandchild будет содержать только один эк- земпляр класса Parent. При выводе мы получим значение переменной, уста- новленное конструктором класса Parent.

## Шаблоны

Еще одно новшество в языке С++, по сравнению с классическим С, — это возможность создания шаблонных функций и классов. Вообще-то *шаблоны* не соответствуют концепции объектно-ориентированного программирования, но оказались настолько удобным механизмом, что практически вся стандарт- ная библиотека построена на шаблонах.

### Шаблон функции

Полезность введения *шаблона функции* рассмотрим на простом примере функции swap(), обменивающей две переменные типа int.

void swap(int& a, int& b)

{

int c = a; a = b;

b = c;

}

Очевидно, что для обмена местами двух переменных типа double необходи- мо построить новую функцию swap(), которая будет отличаться лишь заме- ной имени типа int на double. Вполне естественно, что у разработчиков язы- ка С++ возник соблазн поручить эту работу компилятору. Проблема решается созданием шаблонной функции (листинг 2.17) с помощью следующей конст- рукции:

template <class T>

Описание функции с использованием формального типа T.



Служебное слово class здесь выполняет чисто декоративную функцию для обозначения имени типа и к классам языка С++ не имеет никакого отношения, вместо него допускается использование служебного слова typename.

|  |
| --- |
| **Листинг 2.17. Шаблонная функция swap()** |

template <class T> void swap (T& a, T& b)

{

T tmp = a; a = b;

b = tmp;

}

В программе обратимся к шаблонной функции с параметрами типа int и

double. Убедимся, что все работает правильно.

#include <iostream.h> void main()

{

int i = 1, j = 2; swap (i,j);

cout << i << '\t' << j << endl; double x = 1.1, y = 2.2;

swap (x,y);

cout << x << '\t' << y << endl;

}

На самом деле, компилятор, встретив шаблонную функцию, поступает очень просто, он откладывает ее компиляцию, пока не найдет в тексте программы конкретного обращения к этой функции. Так, встретив обращение swap(i,j), где i, j — переменные типа int, он считает, что T = int и компилирует функцию для типа int. Далее, встретив обращение swap(x,y);, где x,y — пе- ременные типа double, он считает, что T = double и компилирует еще один экземпляр функции для типа double.



Отсюда следует, что шаблон функции и обращение к ней должны находиться в одной единице компиляции. Их нужно описать в одном файле или добавить текст шаблона через файл включений.

Таким образом, для каждого конкретного типа данных строится свой экземп- ляр функции, и эту рутинную работу выполняет компилятор, избавив про- граммиста от необходимости ее проделывать. Причем в описании шаблона может быть перечислено несколько шаблонных типов:

template <class A, class B, . . .>



Если для некоторых значений типа имеется обычная функция, то компилятор не строит шаблонной функции, а использует существующую.

### Шаблон класса

Однако разработчики идеологии шаблона пошли дальше и использовали шаблоны при описании класса. Рассмотрим пример создания класса Stack и посмотрим, как из него можно сделать *шаблонный класс*.

const int N = 100; class Stack

{

private: int st[N];

short top;

public:

};

Stack(): top(-1) { }

void push(int& var) { st[++top] = var; } int& pop() { return st[top--]; }

Класс Stack предназначен для хранения массива целых чисел (до 100). При создании экземпляра класса *стек* пустой и указатель "головы" стека top = -1. Метод push() обеспечивает добавление одного целого числа, он просто уве-

личивает указатель на 1 и помещает число в массив, используя этот указатель как индекс. Таким образом, указатель top всегда показывает на самое верхнее число в стеке. Метод pop() обеспечивает извлечение числа из стека и умень- шает указатель на 1. Реализуется стековый принцип организации хранения данных — *"первым пришел — последним вышел"* (рис. 2.4).



|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |



**Рис. 2.4.** Организация стека

Если теперь нам понадобится создать стек для другого типа данных, напри- мер double, то возникает та же самая проблема — нам придется переписать текст класса, изменив лишь тип данных int на double.

Конечно, гораздо лучше определить класс как шаблонный. Делается это так:

const int N = 100; template <class T> class Stack

{

private: T st[N];

short top;

public:

};

Stack(): top(-1) { }

void push(T& var) { st[++top] = var; } T& pop() { return st[top--]; }

То есть точно так же, как и для шаблона функции, перед описанием класса пишем строку:

template <class T>

и далее используем формальный тип T в описании класса.

Отличием в использовании шаблонных классов от обычных является специ- альный способ описания переменных класса. Дело в том, что мы должны яв- но указать тип данных, для которых создается переменная:

Имя\_класса<тип> переменная, ...;

например:

Stack<int> si;

т. е. после имени класса в угловых скобках указывается тип данных, затем перечисляются переменные. Тестирующая программа для шаблонного класса Stack представлена в листинге 2.18.

**Листинг 2.18. Тестирующая программа для шаблонного класса Stack**

#include "Stack.h" // Подключение шаблонного класса #include "String.h" // Подключение класса String void main()

{// Стек для набора целых чисел Stack<int> si;

for (int i = 0; i < 10; i++) si.push(i);

for (i = 0; i < 10; i++) cout << si.pop() << endl;

// Стек для данных типа String Stack<String> str;

String name[5] = {"Сергей", "Дмитрий", "Антон", "Владимир", "Елена"}; for (i = 0; i < 5; i++) str.push(name[i]);

for (i = 0; i < 5; i++) cout << str.pop()<< endl;

}

Здесь мы построили стек si для данных целого типа и стек str для перемен- ных типа String на одном и том же шаблонном классе.

Вывод на консоль:

9

8

7

6

5

4

3

2

1

0

Елена Владимир Антон Дмитрий Сергей

Обратите внимание на то, что при описании массива типа String мы исполь- зовали возможности конструктора класса String, принимающего в качестве параметра char\*.

Возможности шаблонов этим не ограничиваются, в шаблоне класса можно также описывать константные значения. Так, в нашем примере разработки класса Stack реализована не очень удачная идея — описывать размер стека в самом описании класса. В этом случае для создания стека большего размера пришлось бы изменять текст самого класса. Оказывается, можно ввести опи- сание константной переменной в описание самого шаблона, в нашем случае:

template <class T, int N>

Разумеется, описание константной переменной const int N = 100; нужно удалить, но в этом случае переменная типа стек будет определяться с явным указанием не только типа данных, но и размера стека:

Stack<int, 100> si;

Чтобы понять, почему это возможно, вспомним, что компилятор начинает компилировать шаблонный класс лишь тогда, когда встретит соответствую- щее описание переменной и когда имеется информация о конкретном типе данных и о значении константы. Должно быть ясно, что в качестве константы может быть использовано такое выражение, которое может быть вычислено на этапе компиляции.

#### Описание методов шаблонного класса в отдельном файле

До сих пор мы описывали методы класса прямо в описании класса. Часто так бывает удобнее и проще, но, если методы становятся слишком громоздкими, их описание разумнее вынести в отдельный файл, оставив в описании класса лишь объявления этих методов. Однако описание шаблонных методов имеет некоторую специфику, так описанию каждого метода должно предшество- вать объявление:

template <class T>

а перед именем метода стоять область видимости:

Имя\_класса<шаблоны>::

И, что важно в этом случае, в качестве файла включений нужно использовать не заголовочный файл с описанием класса, а файл реализации методов с рас- ширением cpp, чтобы компилятор имел доступ к тексту шаблонного метода во время компиляции программы. Файлы Stack.h и Stack.cpp приведены в листинге 2.19.

|  |
| --- |
| **Листинг 2.19. Класс Stack с реализацией методов в отдельном файле** |

// Файл Stack.h

template <class T, int N> class Stack

{

private: T st[N];

short top;

public:

};

Stack();

void push(T& var); T& pop();

//////////////////////////////////////////////////

// Файл Stack.cpp template <class T, int N>

Stack<T,N>::Stack(): top(-1) { }

//

template <class T, int N>

void Stack<T,N>::push(T& var) { st[++top] = var; }

//

template <class T, int N>

T& Stack<T,N>::pop() { return st[top--]; }

// В головной функции заменим файл включений

// Файл main.cpp #include "Stack.cpp" #include <iostream> using namespace std; void main()

{

Stack<int, 100> si;

for (int i = 0; i < 10; i++) si.push(i);

for (i = 0; i < 10; i++) cout << si.pop() << endl;

}

## Обработка исключений

*Исключением* или *исключительной ситуацией* (exception) называется преры- вание нормальной работы программы в ответ на непредвиденное или аварий- ное событие. В С++ реализован специальный механизм для обработки ис- ключительных ситуаций. Прежде чем обсуждать, зачем это сделано, посмот- рим, как это сделано.

В самом простом случае, внутри функции в том месте, где возможно возник- новение критической ситуации, ставится блок try, после которого может сто- ять несколько блоков перехватчиков catch, например:

void main()

{ . . .

try

{

...

}

catch(тип\_ошибки)

{

...

}

catch(...)

{

...

}

. . .

}



Блок try называют "охраняемым разделом кода". Иногда в литературе встре- чается название "блок повторных испытаний", что вряд ли является удачным названием, поскольку повторно этот блок управления не получает.

В случае возникновения исключения, а системные функции могут порождать *исключительные ситуации*, поочередно проверяются аргументы блоков, и управление передается тому блоку, тип аргумента которого совпал с типом, порожденным исключением. Если же ни один из типов не совпал, то управ- ление передается блоку со специальным именем (...), который обрабатывает любые исключения (ясно, что этот блок должен стоять последним), если же такого блока нет, исключение обрабатывается системными средствами, что неизбежно приводит к завершению работы программы. Нужно иметь в виду, что модель обработки исключений, реализованная в С++, является *невозоб-*

*новляемой*. При возникновении исключения вначале уничтожаются все *ав- томатические переменные*, созданные в блоке try до точки возникновения исключения, при необходимости вызываются деструкторы. Переменные, соз- данные в куче оператором new, автоматически не уничтожаются. Затем ис- ключение обрабатывается в соответствующем блоке catch, после чего управ- ление передается первому исполняемому оператору после последнего блока catch.



Можно конечно исхитриться, поместив охраняемый раздел внутрь оператора цикла и устанавливая значения логической переменной, или же, доходя до не- приличия, при помощи оператора goto.

Рассмотрим пример обработки исключения (листинг 2.20), возникающего при делении на 0. Это системное исключение или *исключение Win32 API*, и един- ственный способ прямого перехвата подобного исключения является исполь- зование блока catch(...).

|  |
| --- |
| **Листинг 2.20. Пример программы обработки системного исключения** |

#include <iostream> using namespace std; void main()

{

int a = 1, b = 0; try

{

int c = a/b;

}

catch (...)

{

cout << "Division into zero\n";

}

}

Вывод:

Division into zero

Можно и самостоятельно порождать исключения оператором throw. Напри- мер, в предыдущей задаче мы могли бы заранее проверить возможность воз- никновения критической ситуации и самостоятельно породить исключение (листинг 2.21), причем возможно передать в блок catch параметр.

**Листинг 2.21. Программа, самостоятельно порождающая исключение**

#include <iostream> using namespace std; void main()

{

int a = 1, b = 0; try

{

if (b == 0) throw "Division into 0"; int c = a/b;

}

catch (char \*str)

{

cout << str << endl;

}

}

Вывод:

Division into 0

Иногда возникает ситуация, когда мы вынуждены прекратить обработку ошибки и вызвать системный обработчик исключения. Это достигается опе- ратором throw без аргументов в блоке обработчика ошибок. Оператор приво- дит к генерации того же исключения, которое привело в блок catch, и вызы- вает системный обработчик. Так, в нашем случае, если мы добавим в блок catch этот оператор:

catch (char \*str)

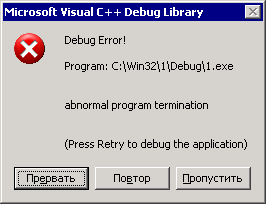
{

throw;

}

Тогда получим системное сообщение (рис. 2.5).

Обычно, однако, так с исключениями не работают, поскольку все рассмот- ренные ситуации можно обработать более простыми средствами. При работе с объектами принято, что исключения порождаются их методами в ответ на некоторые нестандартные ситуации (например, нехватка памяти и т. п.), что не позволяет далее работать программе. Пусть у нас имеется класс, методы которого могут порождать исключительные ситуации. Например, деление на 0, или попытка освобождения несуществующего блока памяти. Опишем внутри этого класса специальный класс ошибок, основное назначение кото- рого заключается в возвращении более детальной информации об ошибке (листинг 2.22).



**Рис. 2.5.** Системное сообщение об ошибке

**Листинг 2.22. Структурная схема класса с возможностью генерации исключения**

class А

{

private:

...

public:

class Error // Описание класса исключений

{

...

};

method()

{

...

throw Error(); // Генерация исключения

}

...

};

Теперь, в том месте программы, где возможно появление ошибки, связанной с классом А, вставим блок try, после которого стоит *блок обработки ошибок* catch:

try

{

...

}

catch(А::Error)

{

...

}

Смысл этого выражения заключается в том, что при возникновении ошибки типа Error класса А управление будет передано блоку обработчика ошибок catch, где происходит ее обработка. Здесь используется своеобразный син- таксис — аргументом catch является тип ошибки, а поскольку класс Error описан внутри класса А, то указывается и область видимости А::Error.

В данном случае мы рассмотрели вариант "пустого" класса ошибок, но воз- можно описание переменных класса Error, через которые и происходит пере- дача дополнительной информации. Рассмотрим обработку ошибок на приме- ре шаблона класса Stack (листинг 2.23).

**Листинг 2.23. Класс с обработкой ошибок**

template <class T, int N > class Stack {

private: T st[N];

short top;

public:

class Range

{

public: char report[20];

// Конструктор служит для передачи сообщения Range(char \*s) { strcpy(report, s); }

};

Stack(): top(-1) { } void push(T& var)

{

if (top >= N-1) throw Range("Stack is full"); st[++top] = var;

}

T& pop()

{

if (top < 0) throw Range("Stack is empty"); return st[top--];

}

};

/////////////////////////////// #include "Stack.h"

#include <iostream> #include <cstring> using namespace std; void main()

{

Stack<int, 100> si;

try

{

for (int i = 0; i < 10; i++) si.push(i);

for (i = 0; i < 11; i++) cout << si.pop() << endl;

}

catch(Stack<int, 100>::Range& e)

{

cout << "Exception: " << e.report << endl;

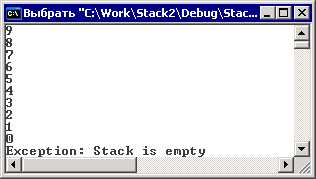
}

}

Здесь, в классе ошибок Range, мы описали символьный массив, через который передадим сообщение Stack is full, если стек заполнен, или Stack is empty, если стек пустой. Таким образом, можно реагировать на возникающие ситуа- ции. Например, в приведенном тексте программы мы намеренно совершили ошибку, пытаясь извлечь из стека 11 значений, тогда как поместили туда все- го лишь 10.

Обратите внимание на то, как мы описали аргумент оператора catch с пере- менной Stack<int, 100>::Range& e. Здесь e — открытая переменная типа Range, которая может использоваться для доступа к полям этого класса.

Вывод программы приведен на рис. 2.6.



**Рис. 2.6.** Окно вывода программы

## Вопросы к главе

1. В чем заключается основной принцип объектно-ориентированного подхо- да к программированию?
2. Синтаксис описания класса.
3. В чем заключается основная идея инкапсуляции данных?
4. Этапы создания нового класса в среде Visual C++ 6.0.
5. Доступ к переменным класса.
6. Конструкторы и деструкторы класса, особенности синтаксиса.
7. Особенности реализации копирующего конструктора.
8. Описание переменных класса, их инициализация.
9. Синтаксис описания методов класса в файле описания класса и в файле реализации.
10. Назначение и особенности статической функции класса.
11. Перегрузка операторов.
12. Необходимость перегрузки оператора присваивания.
13. Особенности реализации оператора преобразования типа.
14. Зачем нужны дружественные функции и операторы?
15. Решение проблемы ввода/вывода кириллического текста на консоль.
16. Объявление пространства имен. Доступ к переменным другого простран- ства имен.
17. Наследование классов, ключевое слово protected.
18. Доступ к методам родительских классов при совпадении имен.
19. Класс, как пользовательский тип данных.
20. Виртуальные методы класса, чисто виртуальные методы. Виртуальные классы.
21. Необходимость использования виртуальных деструкторов.
22. Объявление шаблонной функции.
23. Шаблонные классы, описание переменных.
24. Механизм генерации исключений.
25. Передача информации об исключительной ситуации.

## Задание для самостоятельной работы

1. Для класса Time перегрузить операторы:

 + для добавления к экземпляру класса Time n часов в виде целого числа;  – для вычисления разности между двумя экземплярами класса Time;

 "преобразования типа Time –> double", где часы будут представлять целую часть числа, а минуты — дробную часть (1.0 = 60 мин).

1. Добавить к реализации класса еще одну целую переменную — дни (day). Переопределить все методы и операторы класса.
2. Решить предыдущие задачи с описанием методов в файле реализации.
3. Реализовать собственный класс Date, где дата хранится в виде целого числа, равного количеству дней от 1 января 1900 года. Определить допус- тимый диапазон хранения даты для типа unsigned int.



Високосным считается год, который делится на 4, но не делится на 100, так 1900 год не является високосным. Исключение — год, который делится на 400, снова является високосным, т. е. 2000 год — високосный.

1. Для класса String создать методы:

 count(), подсчитывающий количество слов в текстовой строке. Можно использовать стандартную функцию strtok();

 DoubleToString(double x, int n, int p), преобразующий число в тек- стовую строку:

* + x — число с плавающей точкой;
  + n — размер поля для преобразования;
  + p — количество знаков после десятичной точки.

1. Построить класс-оболочку Integer с единственной закрытой переменной типа int и конструкторами:

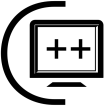
 Integer();

 Integer(int); Integer(char\*);

 Integer(Integer&).

Перегрузить арифметические операторы: +, -, \*, /, %, а также операто- ры консольного ввода/вывода.

1. Создать класс "Числовой вектор" для данных типа double и перегрузить для него основные операции: сложение, вычитание, скалярное произве- дение.
2. Сделать класс "Числовой вектор" шаблонным классом. Построить пример для вектора типа Integer.
3. Для задачи "Решение системы линейных уравнений" из *главы 1* организо- вать генерацию и обработку исключения в случае ее несовместности.
4. Для класса Stack обеспечить обработку переполнения стека. При попытке записи данных в заполненный стек обеспечить увеличение его размера на N.

**Г ЛА В А 3**

# Потоки

Рассмотрим подробнее работу с потоковым вводом/выводом. Если в класси- ческом С ввод/вывод организован с использованием библиотек, размещенных в файлах включений io.h и stdio.h, где связь с файлами осуществлялась че- рез дескрипторы и файловую переменную, то в современном стиле програм- мирования считается предпочтительней использовать потоковый ввод/вывод, в связи с чем мы не стали рассматривать классический подход, а сразу пере- ходим к рассмотрению потоков.

## Консольный ввод/вывод

Имеется четыре предопределенных потоковых объекта, которые становятся доступны при включении в программу заголовочного файла iostream:

* cin — используется для ввода с клавиатуры;
* cout — используется для вывода на экран монитора;
* cerr — используется для вывода сообщений об ошибках;
* clog — используется для ведения журнала.

С объектами cin и cout мы уже знакомы, это экземпляры потокового класса istream и ostream. Теперь рассмотрим их более подробно. Оказывается, име- ется масса возможностей управления вводом/выводом, для этого необходимо лишь выставить определенные флаги форматирования, которые и будут управлять процессом. Можно рассматривать управление вводом/выводом c потоковыми классами из заголовочного файла iostream.h, но лучше сразу перейти на новую библиотеку ввода/вывода, имеющую более богатые воз- можности управления потоками, и использовать заголовочный файл iostream. Новые классы определены в стандартной области имен, поэтому необходимо сделать ее доступной директивой using namespace std;.

### Флаги

*Флаги* служат для управления процессом ввода/вывода и настройки потоко- вых методов, каждый флаг представляет собой один бит слова состояния по- тока. Все они могут быть установлены при помощи метода setf() и сброше- ны методом unsetf(). Символические обозначения флагов размещены в об- ласти имен ios, поэтому для доступа к ним необходимо указывать область видимости ios::. Флаги (табл. 3.1) могут комбинироваться при помощи опе- рации битового сложения.

Приведем пример установки и снятия флагов:

cout.setf(ios::fixed | ios::showpos); cout.unsetf(ios::showpos);

***Таблица 3.1.*** *Флаги форматного ввода/вывода*

|  |  |
| --- | --- |
| **Флаг** | **Описание** |
| skipws | Пропуск разделителей при вводе |
| left | Выравнивание по левому краю |
| right | Выравнивание по правому краю |
| internal | Выравнивание знака по левому краю, а значения — по правому краю |
| boolalpha | Вывод логических величин в текстовом виде |
| dec | Выводить числа в десятичной системе (по умолчанию) |
| oct | Выводить числа в восьмеричной системе |
| hex | Выводить числа в шестнадцатеричной системе счисления |
| showbase | Выводить индикатор основания системы счисления |
| showpoint | Показывать десятичную точку |
| uppercase | Переводить в верхний регистр символы в шестнадцатеричных числах |
| showpos | Показывать знак + для положительных значений |
| scientific | Экспоненциальный вывод чисел с плавающей точкой |
| fixed | Фиксированный вывод чисел с плавающей точкой |
| unitbuf | Сброс потоков после вставки (буферизация вывода) |
| stdio | Сброс стандартных потоков вывода |

### Манипуляторы

Можно управлять потоком при помощи *манипуляторов* (инструкций форма- тирования), которые вставляются прямо в поток (можно для этих целей ис- пользовать и методы потоковых классов). Но для того чтобы иметь возмож- ность использовать манипуляторы, необходимо добавить файл включений iomanip, размещенный в стандартной области имен.

***Таблица 3.2.*** *Манипуляторы потоковых классов*

|  |  |
| --- | --- |
| **Манипулятор** | **Действие** |
| ws | Включает пропуск пробелов при вводе |
| dec | Перевод в десятичную систему |
| oct | Перевод в восьмеричную систему |
| hex | Перевод в шестнадцатеричную систему |
| left | Устанавливает выравнивание по левому краю |
| right | Устанавливает выравнивание по правому краю |
| internal | Устанавливает выравнивание знака по левому краю, а зна- чения — по правому краю |
| endl | Вставка разделителя строк и очистка выходного потока |
| ends | Вставка символа завершения строки '\0' |
| boolalpha | Включает текстовое представление логических данных |
| noboolalpha | Включает числовое представление логических данных |
| showpos | Вывод знака для положительных чисел |
| noshowpos | Вывод положительных чисел без знака |
| uppercase | Вывод символов в числах в верхнем регистре |
| nouppercase | Вывод символов в числах в нижнем регистре |
| showbase | Вывод индикатора основания системы счисления |
| noshowbase | Запрет на вывод индикатора основания системы счисления |
| skipws | Автоматическое игнорирование начальных пропусков при чтении данных оператором >> |
| noskipws | Обработка начальных пропусков при чтении оператором >> |
| unitbuf | Вывод без буферизации |
| nounitbuf | Включение буферизации вывода (по умолчанию) |
| setw(n) | Установить ширину поля вывода в n символов |

***Таблица 3.2*** *(окончание)*

|  |  |
| --- | --- |
| **Манипулятор** | **Действие** |
| setfill(char) | Установить символ-заполнитель |
| setprecision(n) | Количество цифр после десятичной точки |
| setiosflags(fl) | Устанавливает флаги форматирования |
| resetiosflags(fl) | Сбрасывает группу флагов форматирования, аналогично вызову метода setf(0,fl) |
| flush | Очистка выходного потока |

Приведем пример программы с использованием манипуляторов (листинг 3.1).

**Листинг 3.1. Программа с использованием манипуляторов**

#include <iostream> #include <iomanip> using namespace std; void main()

{

int k = 32767;

cout << setw(8) << oct << k << endl; // По основанию 8 cout << setw(4) << hex << k << endl; // По основанию 16 cout << setw(8) << dec << k << endl; // По основанию 10 cout << setprecision(4); // Вывод 4-х знаков

// после десятичной точки

double a = 2., b = 3.;

cout<<"2/3= "<< setfill('\*')<<setw(16)<<internal<<-a/b<<endl;

}

Вывод на консоль:

77777

7fff

32767

2/3= -\*\*\*\*\*\*\*\*\*0.6667

### Методы

Для управления потоками ввода/вывода используются также и *методы*

(табл. 3.3), которые можно разбить на группы.

***Таблица 3.3.*** *Методы управления флагами*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| Flags() | Возвращает все текущие форматные флаги потока. Точнее, возвращает слово состояния флагов форматирования в виде длинного целого числа |
| flags(flags) | Устанавливает все форматные флаги потока в новое состоя- ние, возвращает старое состояние флагов |
| setf(flags) | Устанавливает один или несколько флагов форматирования |
| setf(new,flags) | Сбрасывает группу флагов, указанную вторым аргументом, и устанавливает флаг в новое значение из первого аргумента:  Первый аргумент Второй аргумент  dec, oct, hex basefield left, right, internal adjustfield  scientific, fixed floatfield |
| unsetf(flags) | Сбрасывает флаг форматирования |

Рассмотрим следующий пример манипулирования флагами:

bool b = true;

long old = cout.flags(ios::boolalpha); cout << hex << old << endl;

cout << b << endl; cout.setf(ios::dec,ios::basefield); cout << 1.0/3 << endl;

cout.flags(old); // Восстановление состояния флагов форматирования

Вывод на консоль:

201

true 0.3333

#### Управление полями вывода

Помимо установки флагов и использования манипуляторов, управлять поля- ми вывода можно и при помощи методов класса ostream, которые приведены в табл. 3.4. На самом деле, манипуляторы реализуются при помощи этих ме- тодов, а их текст можно посмотреть в файлах включений.

Пример:

cout.width(10); cout.precision(6);

char old\_fill = cout.fill('\*');

***Таблица 3.4.*** *Методы управления полями потока ostream*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| width() | Возвращает ширину поля вывода |
| width(n) | Устанавливает ширину поля вывода |
| precision() | Возвращает количество знаков после десятичной точки |
| precision(n) | Устанавливает количество знаков после десятичной точки, возвращает старое значение |
| fill() | Возвращает текущий символ заполнения |
| fill(char) | Устанавливает символ заполнения, возвращает старое зна- чение |

#### Методы обмена с потоками класса *istream*

Для чтения данных из входного потока istream ранее мы использовали лишь перегруженный оператор >>, для которого и предназначены рассмотренные выше способы форматирования. Но в классе istream реализована и большая группа методов неформатного ввода, использование которых зачастую быва- ет эффективней, а иногда ввод просто нельзя осуществлять оператором фор- матного ввода, например, при чтении текстовой строки. Все методы приведе- ны в табл. 3.5.

***Таблица 3.5.*** *Методы обмена класса istream*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| >> | Форматированное извлечение данных для всех основных типов из потока |
| gcount() | Возвращает количество символов прочитанных предыдущим методом неформатированного ввода |
| get() | Возвращает код извлеченного из потока символа или EOF(-1) |
| get(ch) | Записывает извлеченный символ в ch |
| get(str,lim='\n') | Записывает в str символы до символа lim, причем вместо символа lim в поток записывается символ '\0', а разделитель lim остается в потоке |
| get(str,MAX,lim='\n') | Записывает в str MAX-1 символов или до символа  lim |
| getline(str,MAX,lim='\n') | То же, что и get(), но символ lim считывается из потока |

***Таблица 3.5*** *(окончание)*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| ignore(MAX=1, lim=EOF) | Пропускает MAX символов или до символа EOF |
| peek() | Читает один символ, оставляя его в потоке |
| putback(ch) | Вставляет последний прочитанный символ обратно в поток |
| read(buf,N) | Считывает N символов в buf, возвращает ссылку на текущий поток |
| readsome(buf,N) | То же, что и read(), но возвращает количество прочитанных символов |

Пример:

#include <iostream> using namespace std; void main()

{

char buf[80], ch; int k;

cout << "Enter string 1: "; cin.get(buf,80);

k = cin.gcount();

cout << "String size = " << k << endl; k = cin.get();

cout << k << endl; cout << buf << endl;

cout << "Enter string 2: "; cin.getline(buf,5); cin.ignore(10,'\n');

cout << buf << endl;

}

Вывод:

Enter string 1: string

String size = 6 Размер введенной строки

10 Символ-разделитель '\n' string

Enter string 2: string

stri В буфер введено только 4 символа

#### Методы обмена с потоками класса *ostream*

Точно так же, как и для класса istream, в классе ostream имеются методы не- форматного вывода данных (табл. 3.6).

***Таблица 3.6.*** *Методы обмена класса ostream*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| << | Форматированный вывод на консоль данных для всех основных типов |
| flush() | Выводит содержимое буфера на консоль, очищает буфер |
| put(ch) | Выводит в поток символ ch |
| write(buf,N) | Записывает в поток N символов из массива buf |



Как и метод flush(), манипулятор endl также обеспечивает вывод строки из буфера и выводит завершающий символ '\n'.

Пример:

char str[] = "text"; cout.write(str, strlen(str)); cout.flush();

## Память как поток

Иногда бывает удобно использовать технологию потокового ввода/вывода в области памяти. Для этого служат 4 потоковых класса, определенных в за- головочном файле strstream:

* istrstream — для чтения из последовательности символов;
* ostrstream — для записи в последовательность символов;
* strstream — класс для чтения и записи;
* strstreambuf — потоковый буфер.

В листинге 3.2 приведен пример, в котором мы прочитаем два целых числа из строки, а затем сформируем строку ввода и выведем ее на консоль.

**Листинг 3.2. Использование памяти как потока ввода/вывода**

#include <strstream> // Заголовочный файл содержит описание #include <iostream> // классов потокового ввода/вывода в память

using namespace std; const int N = 20; void main()

{

int i, j;

char s1[] = "22", s2[] = "33";

// Определяем два входных потока, связанных с символьными строками istrstream in1(s1), in2(s2);

in1 >> i; // Ввод данных in2 >> j;

char buff[N]; // Определение буфера для вывода

ostrstream mem(buff,N); // Определение выходного потока mem << i << '\*' << j << '=' << i\*j << endl << ends;

cout << buff; // Вывод сформированной строки на консоль

}

Вывод:

22\*33=726



Для формирования стандартной С-строки необходимо завершить ее символом

'\0', что можно сделать, используя манипулятор ends.

Используя эту технологию, создадим для рассмотренного ранее класса String методы, которые преобразуют содержимое экземпляра класса в целое число или число с плавающей точкой и, наоборот, из соответствующего числового значения сформируют экземпляр класса String.

Поскольку мы создавали класс String, используя старую библиотеку функ- ций, то, во избежание конфликтов имен, заменим заголовочный файл iostream.h на iostream и добавим директиву разрешения области видимос- ти using namespace std, необходимо также добавить заголовочный файл strstream.

Методы класса String приведены в листинге 3.3.

|  |
| --- |
| **Листинг 3.3. Методы класса String для преобразования числовых данных** |

// Преобразование int -> String void IntToString(int k)

{

if (s) delete[] s; // Удалим предыдущее значение строки char buff[14];

ostrstream mem(buff,14); // Определение выходного потока mem << k << ends;

n = strlen(buff);

s = new char[n]; // Выделим память под новую строку for (int i = 0; i < n; i++) s[i] = buff[i];

}

// Преобразование String -> int int StringToInt()

{

int k;

char \*str = \*this; // Неявное преобразование типа istrstream in(str); // Определение входного потока in >> k;

return k;

}

Два следующих метода построены аналогично:

// Преобразование double -> String void DoubleToString(double q)

{

if (s) delete[] s; char buff[24];

ostrstream mem(buff,24); mem << q << ends;

n = strlen(buff); s = new char[n];

for (int i = 0; i < n; i++) s[i] = buff[i];

}

// Преобразование String -> double double StringToDouble()

{

double q;

char \*str = \*this; istrstream in(str); in >> q;

return q;

}

В силу особенностей работы оператора форматного ввода, в случае некор- ректного задания числового значения в экземпляре String исключительной ситуации не возникает, просто преобразование происходит до первого нечи- слового символа, остаток строки игнорируется. Корректность преобразования необходимо проверять другими средствами.

## Файловый ввод/вывод

В С++ для работы с файлами обычно используют классы потокового вво- да/вывода, определенные в заголовочном файле fstream. Эти классы являют- ся наследниками классов istream и ostream, а значит, наследуют и все их ме- тоды. Однако, учитывая специфику файлов, имеются и специальные методы, например методы произвольного доступа к файлу.

* ifstream — класс входных файловых потоков;
* ofstream — класс выходных файловых потоков;
* fstream — класс двунаправленных файловых потоков.

Открытие файла возможно при помощи конструктора файлового потока. Ес- ли же используется конструктор без параметров, то создается экземпляр фай- лового потока, а связь с файлом будет установлена при помощи метода open(). Закрываются потоки методом close(), но можно положиться на дест- руктор потоковой переменной, который вызовет его автоматически. Для на- глядности приведем все методы для открытия и закрытия файла в табл. 3.7.

***Таблица 3.7.*** *Методы для открытия и закрытия файла*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| open(имя) | Открытие файла потока по умолчанию |
| open(имя,флаги) | Открытие файла потока с установленными флагами |
| close() | Закрытие файлового потока |
| is\_open() | Проверка открытия файла |

Пример создания входного потока из файла, находящегося в текущей папке:

ifstream in("test.txt");

или то же самое, но в 2 приема:

ifstream in; in.open("test.txt");

В качестве второго параметра конструктора указываются флаги, которые за- дают режим открытия файла (табл. 3.8).

***Таблица 3.8.*** *Флаги открытия файла*

|  |  |
| --- | --- |
| **Флаг** | **Описание** |
| in | Открыть для чтения (по умолчанию для ifstream) |
| out | Открыть для записи (по умолчанию для ofstream) |

***Таблица 3.8*** *(окончание)*

|  |  |
| --- | --- |
| **Флаг** | **Описание** |
| ate | Установить указатель на конец файла |
| app | Открыть для добавления в конец файла |
| trunc | Удаление старого содержимого файла |
| binary | Открыть в двоичном режиме |



При открытии файла в текстовом режиме пара символов конца строки "\r\n" при чтении преобразуется в один символ '\n', а при записи — наоборот; при считывании же специального символа "конец файла" чтение файла прекраща- ется. В двоичном режиме никаких преобразований не производится и, если файл не является текстовым, его нужно открывать с флагом binary.

Рассмотрим простой пример чтения текстового файла (листинг 3.4) и вывод содержимого на консоль и в другой файл. Учтем также проблему с выводом на консоль кириллического текста.

**Листинг 3.4. Программа чтения текстового файла**

#include <iostream>

#include <fstream> // Содержит описание классов потоков ввода/вывода #include <windows.h> // Здесь расположен прототип функции CharToOem() using namespace std;

int main()

{

ifstream in; ofstream out;

char name[40], str[82], BufRus[82]; cout << "Input file name: ";

cin >> name; in.open(name);

if (in.fail()) { cout << "Input file not found\n"; return 1; } cout << "Output file name: ";

cin >> name; out.open(name);

if (out.fail()) { cout << "Error in Output file\n"; return 1; } while (in.getline(str,82)) // Читаем строки до конца файла

{

CharToOem(str,BufRus); // Перекодирование кириллицы cout << BufRus << endl;

out << str << endl;

}

in.close();

out.close(); return 0;

}

Вывод:

Input file name: \t.txt Output file name: \t1.txt текст

Здесь мы использовали метод fail() для проверки успешного открытия фай- ла, но это можно сделать и другими средствами, например, проверяя, не яв- ляется ли нулем потоковый объект:

if (!in.open(name)){cout << "Input file not found\n"; return 1;}

Кстати, единственная причина, по которой мы описали функцию main() с возвращаемым значением типа int, заключается в том, что мы можем напи- сать оператор return 1 в случае ошибки открытия файла. В противном слу- чае нам пришлось бы воспользоваться стандартной функцией exit(1).

Состояние потока определяется флагами ошибок, определенными в клас- се ios (табл. 3.9).

***Таблица 3.9.*** *Флаги ошибок потока*

|  |  |
| --- | --- |
| **Флаг** | **Описание** |
| goodbit | Нет ошибок |
| eofbit | Достигнут конец файла |
| failbit | Ошибка форматирования или преобразования |
| badbit | Серьезная ошибка |
| hardfail | Неисправность оборудования |

Имеется целая группа методов проверки состояния потока (табл. 3.10).

***Таблица 3.10.*** *Методы проверки состояния потока*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| rdstate() | Возвращает текущее состояние потока |
| eof() | Возвращает ненулевое значение, если установлен флаг eofbit |

***Таблица 3.10*** *(окончание)*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| fail() | Возвращает ненулевое значение, если установлен один из флагов: failbit, badbit, hardfail |
| bad() | Возвращает ненулевое значение, если установлен один из флагов: badbit, hardfail |
| good() | Возвращает ненулевое значение, если сброшены все флаги ошибок |
| clear(int = 0) | Сбрасывает все флаги и устанавливает указанный в качестве параметра флаг. Если указан без параметра, то сбрасывает все флаги ошибок |

Используя эти методы, можно цикл чтения файла организовать так:

while (in.good()){ ... }

Но если так поступить в предыдущем примере, мы получим в выходном фай- ле и на консоли лишнюю пустую строку, поскольку метод getline() вернет нулевое значение потока уже внутри цикла и произойдет вывод пустой стро- ки, а условие good() проверится лишь в следующей итерации. Кстати, то же самое произойдет, если мы заменим условие на !in.eof(), поэтому мы пред- почтем проверку возвращаемого значения потоковой переменной.

При вводе имени файла нужно учитывать, что если имя вводится без указа- ния пути, то файл будет искаться в текущей папке, если же нужно указать имя файла, находящегося в другой папке, то необходимо указывать либо полное имя файла, либо относительный путь, например:

d:\document\text\test.txt

Файл test.txt находится на диске d в папке text, вложенной в папку

document.

Если опустить имя диска, то по умолчанию понимается текущий диск, если имя начинается с символа \ (слэш), то файл ищется от корневого каталога, иначе поиск осуществляется от текущего каталога. В этом случае можно ис- пользовать относительное имя, например, если текущим каталогом является каталог d:\document, то имя text\test.txt означает тот же самый файл, что мы описали выше.



Здесь термины "каталог" и "папка" являются синонимами, но для описания файловой системы корректнее использовать термин "каталог".

Можно использовать специальные имена:

* . — текущий каталог.
* .. — каталог, расположенный на один уровень выше текущего (родитель- ский каталог).

Так, если мы находимся в каталоге d:\document, то имя ..\test.txt означает, что файл размещается в корневом каталоге диска d.



Если же мы описываем имя файла в тексте программы, необходимо всегда ис- пользовать двойной слэш \\, поскольку это спецсимвол в строке, например: "d:\\document\\text\\test.txt".

### Произвольный доступ к файлам

Суть произвольного доступа заключается в том, что мы можем произвольно перемещать указатель чтения/записи по файлу, не перечитывая все промежу- точные данные. Для этого имеется набор методов для потоков чтения и запи- си, где суффикс "g" означает "get", а суффикс "p" — "put" (табл. 3.11).

***Таблица 3.11.*** *Методы произвольного доступа в потоке*

|  |  |  |
| --- | --- | --- |
| **Класс** | **Метод** | **Действие** |
| ifstream | seekg(pos) | Устанавливает текущую позицию чтения в pos |
| seekg(offs,org) | Перемещает текущую позицию чтения на offs байтов, отсчитывая от одной из трех позиций org: ios::beg, ios::cur, ios::end |
| tellg() | Возвращает текущую позицию чтения потока |
| ofstream | seekp(pos) | Устанавливает текущую позицию записи в pos |
| seekp(offs,org) | Перемещает текущую позицию записи на offs  байтов, отсчитывая от позиции org |
| tellp() | Возвращает текущую позицию записи в поток |

В качестве примера рассмотрим программу, которая выводит в файл массив из 1000 целых чисел в двоичном представлении, затем читает из этого файла лишь каждое 100-е значение и выводит его на консоль (листинг 3.5).

|  |
| --- |
| **Листинг 3.5. Программа с произвольным доступом к файлу** |

#include <iostream> #include <fstream>

using namespace std; int main()

{

int k;

ofstream out("test.dat", ios::binary); for (int i = 0; i < 1000; i++)

out.write((char \*)&i, sizeof(int)); out.close();

ifstream in("test.dat", ios::binary); for (i = 0; i < 1000; i += 100)

{

in.seekg(i\*sizeof(int)); in.read((char \*)&k, sizeof(int)); cout << k << endl;

}

in.close();

}

Вывод:

0

100

200

300

400

500

600

700

800

900

Мы выводим число в файл побайтно следующим методом:

write((char \*)&i, sizeof(int));

где в качестве первого параметра должен стоять указатель типа char\* на об- ласть памяти, содержимое которой выводится, а второй параметр — размер области (у нас используется целое число, занимающее 4 байта). Но поскольку мы выводим данные типа int, то должны поставить явное преобразование int\* -> char\*. Вообще-то в современном стиле программирования принято описывать такое преобразование так:

reinterpret\_cast<char \*>(&i). При вводе, для метода read((char \*)&k, sizeof(int));

ситуация абсолютно симметрична.

Здесь нужно отметить, что чтение из файла и запись в файл по 4 байта не яв- ляется очень плохим решением, поскольку эти операции осуществляются че- рез буфер и по сути дела происходят в оперативной памяти, физическая же операция чтения/записи с диском происходит по мере исчерпания или запол- нения буфера и осуществляется автоматически.

### Доступ к файловому буферу

Операции потокового ввода/вывода, как правило, осуществляются через бу- фер, что повышает производительность системы. Оказывается, можно произ- водить манипуляции с потоком данных непосредственно через буфер. Так, например, можно одной командой вывести все содержимое текстового файла на консоль (пример такой программы приведен в листинге 3.6).

**Листинг 3.6. Простейшая программа вывода текстового файла на консоль**

#include <iostream> #include <fstream> using namespace std; void main()

{

ifstream in("t.txt"); cout << in.rdbuf();

}

Вывод:

string 1

string 2

Здесь метод rdbuf() вернул указатель файлового буфера, а перегруженный оператор << вывел его содержимое на консоль.

Можно также описать переменную типа "указатель файлового буфера" и свя- зать ее с открытым потоком, затем методами потокового буфера манипули- ровать данными. Например, если бы мы захотели в предыдущей задаче вы- вести содержимое файла на консоль посимвольно, можно было бы это сде- лать так:

void main()

{

char ch;

ifstream in("t.txt"); filebuf \*buf = in.rdbuf();

while ((ch = buf->sbumpc()) > 0) cout << ch;

}

Мы описали переменную filebuf \*buf; и присвоили ей значение буфера от- крытого потока, затем методом sbumpc() извлекаем символ за символом из буфера, учитывая, что метод возвращает код символа в виде целого числа, или -1 при достижении конца буфера. Приведем в табл. 3.12 основные мето- ды работы с буфером.

***Таблица 3.12.*** *Методы класса filebuf*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| sputc(ch) | Выводит символ в потоковый буфер |
| sputs(str,n) | Выводит n символов из строки str в потоковый буфер |
| in\_avail() | Возвращает нижнюю границу доступных символов |
| sgetc() | Возвращает текущий символ без его извлечения из буфера |
| sbumpc() | Возвращает текущий символ с извлечением из буфера |
| snextc() | Переходит к следующему символу и возвращает его |
| sgetn(str,n) | Читает n символов и сохраняет их в str |
| sputbackc(ch) | Возвращает символ обратно в потоковый буфер |
| sungetc() | Возвращает указатель текущей позиции к предыдущему сим- волу |

### Итераторы потоковых буферов

Другой механизм непосредственного доступа к буферу потока основан на использовании классов *итераторов* потоковых буферов, которые предназна- чены для посимвольного чтения/записи в буфере потока и являются частью стандартной библиотеки. Эти классы описаны в файле включений iterator и реализованы в виде шаблонов.

**Класс *ostreambuf\_iterator***

Для итераторов буфера выходного потока доступны лишь операции записи, после чего символы, помещенные в буфер, уже не доступны (табл. 3.13).

***Таблица 3.13.*** *Методы класса ostreambuf\_iterator*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| ostreambuf\_iterator<char>(ostream) | Создание итератора потокового буфера вывода для потока ostream |

***Таблица 3.13*** *(окончание)*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| ostreambuf\_iterator<char>(buffer) | Создание итератора потокового буфера вывода для буфера, на который ссыла- ется указатель buffer |
| iter = ch | Записывает символ ch в буфер вызовом функции sputc(ch) |
| \*iter | Фиктивная операция возвращает iter |
| ++iter |
| iter++ |
| failed() | Проверка возможности записи в буфер |

**Класс *istreambuf\_iterator***

Итераторы входного потока, напротив, предназначены лишь для чтения дан- ных. Несколько своеобразно решен вопрос о равенстве входных итераторов: все итераторы конца буфера считаются равными.

***Таблица 3.14.*** *Методы класса istreambuf\_iterator*

|  |  |
| --- | --- |
| **Метод** | **Действие** |
| istreambuf\_iterator<char>() | Создание итератора конца потока |
| istreambuf\_iterator<char>(istream) | Создание итератора потокового буфера ввода для потока istream |
| istreambuf\_iterator<char>(buffer) | Создание итератора потокового буфера ввода для буфера, на который ссылается указатель buffer |
| \*iter | Возвращает текущий символ |
| ++iter | Читает следующий символ и возвращает его позицию |
| iter++ | Читает следующий символ, но возвра- щает позицию предыдущего символа |
| iter1.equal(iter2) | Проверяет равенство двух итераторов |
| iter1 == iter2 | Проверяет на равенство два итератора |
| iter1 != iter2 | Проверяет на неравенство два итератора |

Смысл использования итераторов заключается в том, что мы работаем с ними как с указателями области памяти, а данные, на самом деле, отправляются в

поток вывода при записи или считываются из входного потока при чтении. Передвигаясь же по входному потоку операцией ++, мы просто пропускаем байты входных данных.

Приведем пример программы ввода/вывода с использованием итераторов по- токового буфера в листинге 3.7.

**Листинг 3.7. Программа ввода/вывода с использованием итераторов потокового буфера**

#include <iterator> #include <iostream> #include <fstream> using namespace std; void main()

{// Вывод строки на консоль, итератор связан с потоком char buf[20] = "text"; ostreambuf\_iterator<char> it(cout);

for (int i = 0; buf[i]; i++) \*it = buf[i];

\*it = '\n';

// Вывод на консоль набора цифр, итератор связан с буфером потока ostreambuf\_iterator<char> iter(cout.rdbuf());

for(i = 0; i < 10; i++) \*iter = i + '0', \*iter = ' ';

\*iter = '\n';

// Вывод на консоль содержимого текстового файла,

// итератор связан с входным файловым потоком ifstream file("t.txt");

istreambuf\_iterator<char> end\_it, input\_it(file); while(input\_it != end\_it) \*it = \*input\_it++;

\*it = '\n';

}

Вывод:

text

0 1 2 3 4 5 6 7 8 9

string 1

string 2



Описание входного потока требует задания двух итераторов: istreambuf\_ iterator<char> end\_it, input\_it(file). Первый из них будет образован конструктором по умолчанию и является итератором конца потока, второй ите- ратор связан с потоком и позволяет читать из него данные. Стандартная конст- рукция для чтения всего файла: while(input\_it != end\_it) { ... }.

## Вопросы к главе

1. Предопределенные классы консольного ввода/вывода.
2. Флаги форматирования, их установка и снятие.
3. Манипуляторы потокового ввода/вывода.
4. Методы потокового ввода/вывода.
5. Создание потока ввода/вывода в память.
6. Файловые потоки, открытие потока на ввод и вывод.
7. Организация чтения и записи файла.
8. Методы чтения и записи двоичного файла.
9. Произвольный доступ к файлу.
10. Методы работы с файловым буфером.
11. Итераторы потокового буфера.
12. Чтение и запись файла с помощью итераторов потокового буфера.

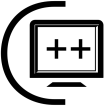
## Задание для самостоятельной работы

1. Написать программу для тестирования работы флагов управления вво- дом/выводом на консоль.
2. Создать аналогичную программу для манипуляторов.
3. Используя управление выводом на консоль, организовать вывод данных типа double:

 в поле шириной 12 позиций со знаком и 6 значащими цифрами после десятичной точки;

 в аналогичном формате вывести таблицу из трех колонок чисел, между колонками обеспечить вывод 4 пробелов.

1. Написать программу, которая вводит массив строк с консоли до пустой строки и выводит его обратно на консоль.
2. Написать программу, которая вводит набор разделенных пробелами чисел в виде строки. Подсчитать количество введенных чисел и, выделив па- мять, записать эти числа в массив типа double, используя память как поток ввода. Вывести результат на консоль.
3. Написать программу, которая читает текстовый файл и подсчитывает ко- личество строк, слов и символов. Результат вывести на консоль.
4. Написать программу, которая принимает строку текста с консоли и до- бавляет к существующему текстовому файлу. Имя файла также вводится с консоли.
5. Описать в программе массив типа double. Вывести массив в двоичном представлении в файл. Затем прочитать этот файл и вывести содержимое на консоль.
6. Создать программу для "склеивания" двух двоичных файлов.
7. Написать программу, которая создает копию текстового файла, используя потоковый буфер.

**Г ЛА В А 4**

# Стандартная библиотека шаблонов STL

*Стандартная библиотека шаблонов* (STL) была разработана Александром Степановым и Менг Ли, сотрудниками фирмы Hewlett-Packard, и с 1998 года входит в стандарт языка С++. STL — это часть стандартной библиотеки, ис- пользуемая для хранения и обработки данных. Основные ее компоненты:

* *контейнеры* — предназначены для хранения данных. Контейнеры по- строены в виде шаблонных классов и могут содержать данные любого ти- па. Основная идея организации хранения данных в контейнерах заключа- ется в том, чтобы избавить пользователя от необходимости заботиться о выделении памяти и обеспечить удобными средствами манипуляции данными. STL включает в себя семь основных типов контейнеров и три производных. Различают две категории контейнеров:

 *последовательные* — векторы, списки, очереди с двусторонним досту- пом (vector, list, deque);

 *ассоциативные* — множества, мультимножества, отображения и муль- тиотображения (set, multiset, map, multimap).

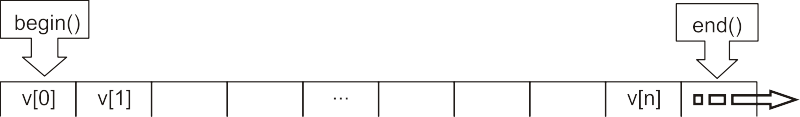
Кроме того, наследниками последовательных контейнеров выступают стек, очередь и приоритетная очередь (stack, queue, priority queue);

* *алгоритмы* — процедуры, применяемые к контейнеру для обработки дан- ных. Они максимально унифицированы с целью обеспечения возможности работы с большинством контейнеров, включая обычные массивы данных, по сути являющимися простейшими контейнерами;
* *итераторы* — интеллектуальные указатели, они обеспечивают ссылки на элементы контейнера, скрывая от пользователя особенности строения конкретного контейнера.

**Контейнер *Vector***

*Вектор* — это самый простой контейнер, относящийся к классу упорядочен- ных коллекций и реализованный в памяти в виде динамического массива, элементы которого хранятся в определенном порядке и занимают непрерыв- ную область памяти с возможностью расширения с конца. В силу такой орга- низации vector обеспечивает произвольный доступ к своим элементам. Ите- раторы вектора являются итераторами произвольного доступа (т. е. позволя- ют передвигаться по контейнеру в обоих направлениях на произвольное количество элементов), поэтому к вектору применимы все алгоритмы STL (рис. 4.1).

Операции вставки и удаления элементов в конце вектора происходят с высо- ким быстродействием, однако при их выполнении внутри вектора быстро- действие существенно снижается, поскольку все элементы в последующих позициях приходится перемещать на новое место.



**Рис. 4.1.** Структура контейнера vector

При работе с контейнерами вместо указателей используются итераторы, вы- полняющие те же функции, но организованные в виде встроенных классов, что позволяет обеспечить корректную работу со всеми контейнерами. Для итераторов перегружены основные операции, имитирующие их поведение как обычных указателей. Одной из особенностей итераторов является согла- шение, по которому, при указании интервала, итератор конца показывает не на последний элемент, а на элемент, лежащий после последнего, даже если физически таковой отсутствует. Такой интервал называется *полуоткрытым* и обозначается: [begin,end).

Чтобы использовать *вектор* в программе, необходимо подключить заголо- вочный файл:

#include <vector>

Описание класса vector, как и всей стандартной библиотеки, лежит в стан- дартной области имен std.

В программе же, при описании переменной типа vector, необходимо конкре- тизировать тип хранимых данных, например: vector<int> v.

### Операции с векторами

Операции с векторами рассмотрим по группам, составив соответствующие таблицы: конструкторы, немодифицирующие операции, операции сравнения, операции присваивания, доступ к элементам вектора, итераторы вектора, вставка и удаление элементов (табл. 4.1—4.7).

***Таблица 4.1.*** *Конструкторы*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| vector<T> v | Создает пустой вектор |
| vector<T> v(n) | Создает вектор из n элементов |
| vector<T> v2(v1) | Создает копию вектора v1 |
| vector<T> v2(n,elem) | Создает вектор из n копий elem |
| vector<T> v2(begin,end) | Создает вектор из элементов интервала [begin,end) любого контейнера, в том числе и из элементов про- стого массива |

Здесь T — тип данных элементов вектора.

***Таблица 4.2.*** *Немодифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| v.size() | Возвращает размер вектора |
| v.empty() | Возвращает true, если вектор пуст |
| v.max\_size() | Возвращает максимальный размер вектора |
| v.capacity() | Возвращает максимально доступный размер вектора без пере- распределения памяти |
| v.reserve() | Увеличивает емкость вектора, если текущая емкость меньше |

В листинге 4.1 приведена простейшая программа с использованием контей- нера vector.

|  |
| --- |
| **Листинг 4.1** |

#include <iostream> // Файлы включений #include <vector>

using namespace std; void main()

{

int m[] = {1, 5, 3, 4, 0, 2, 1, 4, 8, 3 };

vector <int> v1; // Конструктор по умолчанию vector <int> v2(20); // Вектор из 20 пустых значений vector <int> v3(10,1); // Вектор из 10 единиц

vector <int> v4(v3); // Копия вектора v3

vector <int> v5(m, m+10); // Вектор из 10 элементов массива cout << "size v5 = "<< v5.size() << endl;

cout << "empty v5 = "<< v5.empty() << endl;

cout << "max\_size v5 = "<< v5.max\_size() << endl; cout << "capacity v2 = "<< v2.capacity() << endl; v5.reserve(20);

}

Вывод:

size v5 = 10 empty v5 = 0

max\_size v5 = 1073741823 capacity v2 = 20

***Таблица 4.3.*** *Операции сравнения*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| v1 == v2 | Проверяет равенство двух векторов |
| v1 != v2 | Проверяет неравенство двух векторов |
| v1 < v2 | Проверяет, что v1 < v2 |
| v1 > v2 | Проверяет, что v1 > v2 |
| v1 <= v2 | Проверяет, что v1 <= v2 |
| v1 >= v2 | Проверяет, что v1 >= v2 |

Чтобы протестировать операции сравнения, добавим к тексту программы из листинга 4.1 следующие три строки:

if (v3 == v4) cout << "v3 is equal v4\n";

vector <int> v6(m, m+3); // Вектор из 3-х элементов массива if (v6 < v5) cout << "v6 < v5\n";

Тогда и в выводе программы прибавятся строки:

v3 is equal v4 v6 < v5

***Таблица 4.4.*** *Операции присваивания*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| v1 = v2 | Присваивает v1 все элементы v2 |
| v.assign(n,elem) | Присваивает n копий elem |
| v.assign(beg,end) | Присваивает элементы интервала [beg,end) |
| v1.swap(v2) | Меняет местами содержимое v1 и v2 |
| swap(v1,v2) | То же в форме функции |

Для демонстрации работы операций присваивания добавим еще несколько строк к нашей программе (листинг 4.1):

v1 = v6;

for (int i = 0; i < v1.size(); i++) cout << v1[i] << endl; v2.assign(100,0);

cout << "size v2 = "<< v2.size() << endl;

cout << "capacity v2 = "<< v2.capacity() << endl; v6.swap(v5);

cout << "size v6 = "<< v6.size() << endl; cout << "size v5 = "<< v5.size() << endl;

Вывод программы для этих операций:

1

5

3

size v2 = 100 capacity v2 = 100 size v6 = 10 size v5 = 3

Некоторые из перечисленных в табл. 4.1—4.4 операций можно прокомменти- ровать следующим образом:

* в операторе присваивания v1 = v6, несмотря на то, что переменной v1 изначально не было выделено памяти для хранения данных, контейнер сам выделил необходимую память, куда и скопировал содержимое перемен- ной v6;
* метод assign() также выделил необходимую память для вектора v2;
* а вот метод swap() обменял не только данные двух векторов, но и их раз- меры.

***Таблица 4.5.*** *Доступ к элементам вектора*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| v[index] | Возвращает элемент с индексом index |
| v.at(index) | Возвращает элемент с индексом index. Генерируется исключение  out\_of\_range при выходе за границы интервала |
| v.front() | Возвращает первый элемент |
| v.back() | Возвращает последний элемент |

Мы уже немного забежали вперед, используя обращение к элементу вектора, как к элементу обычного массива v1[i]. Это действительно допустимо, и с вектором можно обращаться как с массивом данных, однако имеются и до- полнительные возможности. Так, метод at(), возвращая элемент контейнера, обеспечивает контроль выхода индекса за границы допустимого интервала, он порождает исключение при выходе за границу интервала, остальные мето- ды в этой ситуации, как правило, приводят к ошибке времени выполнения. Например, если мы добавим к нашему примеру строку:

for (i = 0; i < 4; i++) v5[i] = 1;

то получим ошибку времени выполнения, поскольку размер вектора v5 сей- час равен 3.

Методы front() и back() возвращают первый и последний элементы век- тора.

***Таблица 4.6.*** *Итераторы вектора*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| v.begin() | Возвращает итератор произвольного доступа первого элемента |
| v.end() | Возвращает итератор произвольного доступа за последним эле- ментом |
| v.rbegin() | Возвращает обратный итератор первого элемента для перебора в обратном направлении |
| v.rend() | Возвращает обратный итератор последнего элемента для перебо- ра в обратном направлении |

Разработчиками STL приняты соглашения, согласно которым при указании диапазона начальный итератор показывает на первый элемент диапазона, а конечный — на элемент, следующий за последним элементом диапазона. В связи с этим достаточно оригинально решена проблема продвижения по контейнеру в обратном направлении: итератор начала показывает на послед-

ний элемент, а итератор конца — на фиктивный элемент, стоящий перед на- чальным. Их можно было бы назвать "начало конца вектора" и "конец начала вектора".

С использованием итераторов можно обеспечить вывод вектора на консоль с помощью следующей конструкции:

vector<int>::iterator it; // Описание итератора

for (it = v5.begin(); it < v5.end(); it++) cout << \*it << '\t';

Вывод:

1 5 3

Здесь конструкция vector<int>::iterator it обеспечивает описание итерато- ра it для вектора с данными типа int, и в ней потребовалось явно указать имя контейнера с уточнением типа данных. Доступ же к данным возможен при- менением обычной операции разадресации итератора it. Далее, установив итератор на начало вектора, it = v5.begin(), передвигаемся по вектору, на- ращивая итератор it++, до достижения конца вектора it < v5.end().

\*it — это значение элемента вектора (операция разадресации для итератора перегружена и имеет тот же смысл, что и для указателя).

***Таблица 4.7.*** *Вставка и удаление элементов*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| v.insert(pos,elem) | Вставляет в позицию pos элемент, возвращает позицию следующего элемента |
| v.insert(pos,n,elem) | Вставляет n элементов elem в позицию pos |
| v.insert(pos,beg,end) | Вставляет элементы интервала [beg,end) в по- зицию pos |
| v.push\_back(elem) | Вставляет элемент elem в конец вектора |
| v.pop\_back() | Удаляет последний элемент |
| v.erase(pos) | Удаляет элемент в позиции pos и возвращает позицию следующего элемента |
| v.erase(beg,end) | Удаляет все элементы из интервала [beg,end) |
| v.resize(num) | Приводит контейнер к размеру num |
| v.resize(num,elem) | Приводит контейнер к размеру num, новые эле- менты создаются как копии elem |
| v.clear() | Удаляет содержимое вектора |

В листинге 4.2 приведен текст тестовой программы для демонстрации работы перечисленных в табл. 4.1—4.7 методов.

|  |
| --- |
| **Листинг 4.2** |

#include <iostream> using namespace std; void main()

{

vector <int> v;

// Заполнение вектора

for (int i = 0; i < 10; i++) v.push\_back(i); while (!v.empty()) // Пока вектор не пустой

{cout << v.back() << ' '; // Вывод на консоль

// последнего элемента

v.pop\_back(); // Его удаление

}

cout << endl;

vector <int> q(5,2); // Вектор из пяти двоек q.insert(q.begin(),5,0); // Вставить первые 5 нулей q.insert(q.begin()+5,5,1); // Вставить следующие 5 единиц for (i = 0; i< 15; i++) cout << q[i] << ' ';

cout << endl;

q.erase(q.begin()+5,q.begin()+10); // Удалить 5 единиц for (i = 0; i< 10; i++) cout << q[i] << ' ';

/\* struct Plane {}

string planeName;

int planeNumber;

Plane\* prev;

Plane\* next; };

Вывод:

9 8 7 6 5 4 3 2 1 0

0 0 0 0 0 1 1 1 1 1 2 2 2 2 2

0 0 0 0 0 2 2 2 2 2

size q = 5 empty q = 1

При работе с контейнерами STL нужно иметь в виду, что при совершении операций вставки элемента или увеличения размера контейнера может авто- матически происходить перераспределение памяти, т. е. для данных выделя-

ется новая область памяти, куда они и копируются, а "старая" область памяти освобождается. Это приводит к тому, что после таких операций необходимо заново определять итераторы начала и конца контейнера.

## Алгоритмы

*Алгоритмы* — это рабочие лошади STL, они выполняют операции над эле- ментами контейнера. Основная идея в разделении реализации контейнеров и алгоритмов заключалась в том, что алгоритмы должны работать с макси- мально возможным числом контейнеров. По способу реализации алгоритмы являются шаблонными функциями, их описание размещено в файле включе- ний algorithm в стандартной области имен std.

При описании алгоритмов мы будем приводить их прототипы, чтобы можно было разобраться в количестве и типе параметров. Как правило, алгоритмы принимают в качестве параметров интервал, заданный итераторами, и пара- метры, задающие режим работы. Причем, все алгоритмы STL работают с обычными массивами и принимают указатели для задания интервала, однако, как и для контейнеров, концом интервала является указатель элемента масси- ва после последнего. Список основных алгоритмов представлен в табл. 4.8.

***Таблица 4.8.*** *Список основных алгоритмов*

|  |  |
| --- | --- |
| **Алгоритм** | **Назначение** |
| find() | Возвращает итератор первого элемента с указанным значением |
| for\_each() | Выполняет указанную функцию для каждого элемента контей- нера |
| count() | Считает количество элементов, имеющих указанное значение |
| equal() | Сравнивает содержимое двух контейнеров |
| search() | Ищет совпадение последовательности из одного контейнера в другом |
| copy() | Копирует последовательность значений из одного контейнера в другой |
| swap() | Обменивает значения |
| iter\_swap() | Обменивает последовательность значений |
| fill() | Заполняет последовательность значений |
| sort() | Сортирует значения в указанном порядке |
| merge() | Совмещает два отсортированных диапазона |
| accumulate() | Возвращает сумму элементов в указанном диапазоне |

***Таблица 4.8*** *(окончание)*

|  |  |
| --- | --- |
| **Алгоритм** | **Назначение** |
| min\_element() | Возвращает элемент с минимальным значением |
| max\_element() | Возвращает элемент с максимальным значением |
| transform() | Модифицирует элементы |
| generate() | Заменяет элементы результатом операции |
| replace() | Заменяет элемент другим |
| remove() | Удаляет элементы с заданным значением |
| unique() | Удаляет смежные дубликаты |
| reverse() | Переставляет элементы в обратном порядке |
| rotate() | Производит циклический сдвиг элементов |
| partition() | Изменяет порядок следования элементов, чтобы спереди оказа- лись элементы, соответствующие критерию |

### Алгоритмы поиска

Рассмотрим подробнее алгоритмы поиска. Все они возвращают итератор, указывающий на найденное значение, или итератор конца интервала, если требуемое значение не найдено.

**Поиск первого совпадающего элемента. Алгоритмы *find()*, *find\_if()***

Iterator find (Iterator beg, Iterator end, const T& value) Iterator find\_if (Iterator beg, Iterator end, UnaryPredicate op)

Первая форма алгоритма принимает в качестве параметров интервал и кон- стантное значение.

Вторая форма также принимает интервал, а в качестве третьего параметра — функциональный объект с одним параметром.

В листинге 4.3 приведен пример программы, в которой использован алгоритм поиска на примере решения задачи поиска в числовом массиве. В качестве итераторов используются обычные указатели.

|  |
| --- |
| **Листинг 4.3. Поиск в числовом массиве** |

#include <iostream> #include <algorithm>

using namespace std; void main()

{

int arr[] = { 4,7,1,6,8,2,6,4,9,3 };

int \*p = find(arr, arr+10, 2);// Ищем первое вхождение числа 2

/\* Plane\* head = NULL;

void insertInOrder(string name, int number)

Вывод:

Number 2 in a position: 5

Необходимо пояснить, что для поиска номера позиции найденного значения мы воспользовались разностью двух указателей: p — arr, что вполне до- пустимо, поскольку числовой массив занимает непрерывную область памяти.

#### Функциональные объекты

Понятие *функционального объекта* очень широко используется в STL для настройки существующих алгоритмов. Функциональный объект *(преди- кат)* — это объект, который ведет себя как функция, возвращающая логиче- ское значение true или false, и может быть:

* функцией;
* предопределенным функциональным объектом;
* специальным классом, имеющим оператор().



В описании стандартной библиотеки предикатом часто называют функциональ- ный объект с любым типом возвращаемого значения, хотя, строго говоря, пре- дикат должен возвращать логическое значение.

Рассмотрим вначале наиболее простой случай, когда функциональный объект является просто функцией. Поставим следующую задачу: необходимо найти для предыдущего примера, приведенного в листинге 4.3, позицию элемента со значением больше 7. Текст программы приведен в листинге 4.4.

|  |
| --- |
| **Листинг 4.4. Использование функции в качестве функционального объекта** |

/\* Plane\* newNode = new Plane;

newNode->planeName = name;

newNode->planeNumber = number;

{

int arr[] = { 4,7,1,6,8,2,6,4,9,3 };

int \*p = find\_if(arr, arr+10, greater);

cout << "Number 2 in a position: " << p - arr << endl;

}

Вывод:

Number 2 in a position: 4

Действительно, число 8 занимает четвертую позицию в массиве.

Обратите внимание, что имя функции greater в алгоритме find\_if() исполь- зуется без аргументов, по сути — это просто передача указателя на функцию, а задача построить правильное обращение к этой функции возлагается пол- ностью на алгоритм.

##### Предопределенные функциональные объекты

Предопределенные функциональные объекты (табл. 4.9) размещены в заголо- вочном файле functional стандартной области имен и реализованы в виде шаблонных структур.

***Таблица 4.9.*** *Предопределенные функциональные объекты*

|  |  |
| --- | --- |
| **Функциональный объект** | **Операция** |
| negate<>() | -x |
| plus<>() | x + y |
| minus<>() | x - y |
| multiplies<>() | x \* y |
| divides<>() | x / y |
| modulus<>() | x % y |
| equal\_to<>() | x == y |
| not\_equal\_to<>() | x != y |
| greater<>() | x > y |
| less<>() | x < y |
| greater\_equal<>() | x >= y |
| less\_equal<>() | x <= y |
| logical\_and<>() | x && y |
| logical\_or<>() | x || y |
| logical\_not<>() | !x |

Можно было бы решить предыдущую задачу при помощи стандартного объ- екта greater<>(), но проблема заключается в том, что это бинарный объект, а по формату алгоритма требуется унарный функциональный объект. Для разрешения подобных коллизий используются так называемые *функциональ- ные адаптеры*, при помощи которых бинарный объект можно преобразовать в унарный. Так, выражение:

bind2nd(greater<int>(),7);

позволит решить эту задачу.

В листинге 4.5 приведен пример тестирующей программы.

|  |
| --- |
| **Листинг 4.5** |

#include <iostream> #include <algorithm> #include <functional> using namespace std; void main()

{

if (head == NULL) { head = newNode; /\*

newNode->prev = NULL;

newNode->next = NULL;}

} else {

Результат работы программы будет тем же.

##### Функциональный объект — класс

В общем случае, функциональный объект (предикат) — это класс, который должен иметь в своем составе оператор ():

class FunctionObject { необязательная часть описания класса public:

тип operator () (параметры)

{

тело оператора;

return возвращаемое\_значение;

}

};

Чтобы понять, как будет работать функциональный объект, нужно иметь в виду, что алгоритмы для каждого элемента контейнера строят обращения к этому объекту следующим образом: op(\*it), где it — итератор, последова- тельно извлекающий элементы контейнера.

Текст программы, в которой показан пример использования класса в качестве функционального объекта, приведен в листинге 4.6.

**Листинг 4.6**

#include <iostream> #include <algorithm> #include <functional> using namespace std; class FunctionObject { private: int val; public:

Plane\* current = head;/\*

Plane\* prev = NULL;

};

void main()

{

int arr[] = { 4,7,1,6,8,2,6,4,9,3 };

FunctionObject op(7);

int \*p = find\_if(arr, arr+10, op);

cout << "Number >7 in a position: " << p - arr << endl; op = 9;

/\* while (current != NULL && current->planeNumber < number) { prev = current; current = current->next;)}

}

Вывод на консоль:

Number >7 in a position: 4

Number >9 in a position: 10

При описании класса мы создали закрытую переменную, которую можем ус- танавливать при помощи конструктора. Теперь, как видно из текста програм- мы, нам достаточно изменить значение переменной op класса FunctionObject, и алгоритм поиска будет работать в другом режиме. Но, поскольку в массиве отсутствуют числа больше 9, поиск завершился возвращением указателя за концом массива.

Теперь, чтобы посмотреть, как алгоритм поиска будет работать с контейне- ром vector, перепишем функцию main(), добавив создание вектора из наше- го массива.

...

#include <vector> void main()

{ int arr[] = { 4,7,1,6,8,2,6,4,9,3 };

// Вектор создается из массива vector<int> v(arr, arr+10); vector<int>::iterator p, beg, end;

beg = v.begin(); // Определяем итератор начала вектора end = v.end(); // и его конца

FunctionObject op(7);// Определение функционального объекта p = find\_if(beg, end, op);

cout << "Number >7 in a position: " << p - beg << endl;

op = 9; // Новое значение функционального объекта p = find\_if(beg, end, op);

cout << "Number >9 in a position: " << p - beg << endl;

}

Здесь нам пришлось определить три итератора и присвоить двум из них зна- чение начала и конца контейнера:

/\* if (prev == NULL) { newNode->prev = NULL; newNode->next = head; head->prev = newNode; head = newNode; }

Как и ожидалось, результат получаем тот же самый.

Обратите внимание на то, как мы определили позицию найденного значения: p - beg. Просто нашли разность итераторов найденного числа и начала кон- тейнера, т. е. поступили как с обычными указателями. Здесь выполнение этой операции возможно, поскольку вектор занимает непрерывную область памя- ти, но это, пожалуй, единственный контейнер, где расстояние в диапазоне можно определить таким простым способом. Во всех других контейнерах для решения такой задачи нам придется воспользоваться специальной функцией.

#### Минимум и максимум.

**Алгоритмы *min\_element()*, *max\_element()***

Iterator min\_element(Iterator beg, Iterator end)

Iterator min\_element(Iterator beg, Iterator end, BinaryPredicate op) Iterator max\_element(Iterator beg, Iterator end)

Iterator max\_element(Iterator beg, Iterator end, BinaryPredicate op)

Все версии только что перечисленных алгоритмов возвращают итератор най- денного элемента. Если таких элементов несколько, алгоритмы возвращают первый найденный элемент. Алгоритмы с двумя аргументами используют

для сравнения оператор <, а версии с тремя аргументами используют для сравнения функциональный объект. Операция op(elem1, elem2) должна воз- вращать true, если elem1 < elem2.

Для проверки работы этих алгоритмов добавим в конец последней програм- мы (см. листинг 4.6 с последующими дополнениями) несколько строк.

/\* else { prev->next = newNode; newNode->prev = prev; newNode->next = current; if (current != NULL) {current->prev = newNode;}}}}

Вывод:

min element= 1

max element= 9

#### Поиск первых *n* последовательных совпадений. Алгоритм *search\_n()*

Iterator search\_n (Iterator beg, Iterator end, Size n, const T& v) Iterator search\_n (Iterator beg, Iterator end, Size n, const T& v,

BinaryPredicate op)

Алгоритм возвращает позицию первого из n последовательных элементов в интервале [beg,end), каждый из которых имеет значение v, или для которых бинарный предикат op(elem,v) возвращает значение true. Если подходящий элемент не найден, обе формы возвращают end.

Добавим в предыдущую программу две строки:

p = search\_n(beg, end, 3, 3, greater<int>());

cout << "Position of three elements big 3 = " << p-beg << endl;

Вывод:

Position of three elements big 3 = 6

#### Поиск первого подинтервала. Алгоритм *search()*

Iterator search (Iterator beg, Iterator end, Iterator searchBeg, Iterator searchEnd)

Iterator search (Iterator beg, Iterator end, Iterator searchBeg, Iterator searchEnd, BinaryPredicate op)

Возвращает позицию первого вхождения в первый интервал [beg,end) иско- мого подинтервала [searchBeg, searchEnd). Во второй форме записи алго- ритма значение true бинарного предиката op(elem, searchElem) служит кри-

терием сравнения. Если подинтервал не найден, обе разновидности алгорит- ма поиска подинтервала возвращают итератор конца интервала end.

Для проверки работы алгоритма создадим вектор и найдем позицию его вхо- ждения в исходный вектор:

/\* void printList() {Plane\* current = head; while (current != NULL) { cout << current->planeName << " - " << current->planeNumber << endl; current = current->next;}}

Вывод:

Position on interval = 5

**Поиск последнего подинтервала. Алгоритм *find\_end ()***

Iterator find\_end (Iterator beg, Iterator end, Iterator searchBeg, Iterator searchEnd)

Iterator find\_end (Iterator beg, Iterator end, Iterator searchBeg, Iterator searchEnd, BinaryPredicate op)

Возвращает позицию последнего вхождения в первый интервал [beg,end) искомого подинтервала [searchBeg, searchEnd). Во второй форме записи алгоритма значение true бинарного предиката op(elem, searchElem) служит критерием сравнения. Если подинтервал не найден, обе разновидности алго- ритма возвращают итератор конца интервала end.

Алгоритм работает практически идентично предыдущему search(), только ищет последнее вхождение подинтервала.

**Поиск первого из нескольких возможных значений. Алгоритм *find\_first\_of()***

Iterator find\_first\_of (Iterator beg, Iterator end, Iterator searchBeg,

Iterator searchEnd)

Iterator find\_first\_of (Iterator beg, Iterator end, Iterator searchBeg,

Iterator searchEnd, BinaryPredicate op) Возвращает позицию первого элемента в интервале [beg,end), значение кото- рого также встречается в интервале [searchBeg, searchEnd), или для которых

бинарный предикат op(elem, searchElem) возвращает значение true. Если

подходящий элемент не найден, обе разновидности алгоритма возвра- щают end.

Например, если нас интересует появление любого из чисел массива:

int m[] = { 2,6,4 };

в исходном массиве мы вставим в программу такой код:

p = find\_first\_of(beg, end, q.begin(), q.end()); cout << "Position = " << p-beg << endl;

Вывод:

Position = 0

**Поиск двух смежных элементов с равными значениями. Алгоритм *adjacent\_find ()***

Iterator adjacent\_find (Iterator beg, Iterator end)

Iterator adjacent\_find (Iterator beg, Iterator end, BinaryPredicate op)

Возвращает итератор первого элемента со значением следующего элемента или же итератор первого из двух элементов, для которых бинарный предикат op(elem, next\_elem) возвращает значение true. Если подходящий элемент не найден, обе разновидности алгоритма возвращают end.

Вставим в вектор значение 1 в позицию beg + 2, и попытаемся найти пару единиц с помощью алгоритма adjacent\_find(). Здесь нас ожидает небольшая неприятность, дело в том, что при вставке элемента вектора может произойти перераспределение памяти, и для корректной работы программы необходимо переопределить итераторы beg и end.

/\*int main(){

insertInOrder("Boeing", 3);

insertInOrder("Airbus", 1); insertInOrder("Embraer", 4);

insertInOrder("Bombardier", 2);

insertInOrder("Cessna", 5);

Вывод:

4 7 1 1 6 8 2 6 4 9 3

/\*printList(); return 0;}

**Алгоритм *for\_each()***

UnaryPredicate for\_each(Iterator beg, Iterator end, UnaryPredicate op)

Алгоритм вызывает унарный предикат op(elem) для каждого элемента в ин- тервале [beg,end). Операция op не может модифицировать элементы, возвра- щаемое значение игнорируется.

Например, для предыдущей задачи мы могли бы организовать вывод элемен- тов вектора на консоль с помощью алгоритма for\_each(). Для этого необхо- димо написать функцию вывода одного элемента вектора:

void print(int elem) { cout << elem << ' '; }

Теперь вставим в программу следующую строку:

for\_each(beg, end, print);

Вывод:

4 7 1 1 6 8 2 6 4 9 3

**Подсчет количества элементов. Алгоритмы *count()*, *count\_if()***

difference\_type cout (Iterator beg, Iterator end, const T& value) difference\_type cout\_if(Iterator beg, Iterator end, UnaryPredicate op)

Алгоритм подсчитывает элементы со значением value в интервале [beg,end), или элементы, для которых унарный предикат op(elem) возвращает значение true. Возвращаемое значение difference\_type соответствует типу разности итераторов (как правило, long).

Например, так мы подсчитаем количество элементов, равных 6, в нашем век- торе:

cout << "count(6) = " << count(beg, end, 6) << endl;

Вывод:

count(6) = 2

#### Проверка на равенство интервалов. Алгоритм *equal()*

bool equal (Iterator beg, Iterator end, Iterator cmpBeg)

bool equal (Iterator beg,Iterator end,Iterator cmpBeg,BinaryPredicate op)

Алгоритм проверяет полное совпадение элементов интервалов [beg,end) и [cmpBeg,...)и возвращает логическое значение. Бинарный предикат op(elem, cmpElem) производит сравнение соответствующих элементов двух интер- валов.

Например, если сравнить два вектора в нашем примере:

cout << "v == q? = " << equal(beg, end, q.begin()) << endl;

Вывод:

v == q? = 0 // false

**Поиск первого несовпадения. Алгоритм *mismatch()***

pair<Iterator1, Iterator2> mismatch(Iterator1 beg, Iterator1 end,

Iterator2 cmpBeg) pair<Iterator1, Iterator2> mismatch(Iterator1 beg, Iterator1 end,

Iterator2 cmpBeg, BinaryPredicate op)

Алгоритм возвращает пару итераторов элементов двух контейнеров при на- хождении первого несовпадения. Если различия не найдены, возвращается один из итераторов end и соответствующий элемент второго контейнера.

В качестве возвращаемого значения используется шаблонная структура пары (pair), которая имеет две открытых переменных first и second. Эта структура часто используется в STL, а ее текст приведем в листинге 4.7.

**Листинг 4.7. Код структуры pair, определенной в заголовочном файле utility**

template<class T1, class T2> struct pair {

T1 first; T2 second;

// Конструктор по умолчанию

pair(): first(T1()), second(T2()) {}

// Конструктор с двумя параметрами

pair(const T1& a, const T2& b): first(a), second(b) {}

// Копирующий конструктор с автоматическим преобразованием типа template<class U, class V>

pair(const pair<U, V> &p): first(p.first), second(p.second) {}

};

Посмотрим сейчас, как изменилось содержимое вектора после вставки эле- мента:

// Объявление переменной типа пары pair<vector<int>::iterator, int \*> val; val = mismatch(beg, end, arr);

cout << "Position vector = " << val.first - beg << endl; cout << "Position array = " << val.second - arr << endl;

Вывод:

Position vector = 3 Position array = 3

**Сравнение по лексикографическому критерию. Алгоритм *lexicographical\_compare()***

bool lexicographical\_compare(Iterator beg1,Iterator end1,Iterator beg2, Iterator end2)

bool lexicographical\_compare(Iterator beg1,Iterator end1,Iterator beg2,

Iterator end2, BinaryPredicate op)

Алгоритм проверяет, меньше ли содержимое интервала [beg1,end1) и интер- вала [beg2,end2) по лексикографическому критерию. Интервалы сравнивают-

ся по значениям соответствующих элементов, и результат этого сравнения является результатом действия алгоритма. Если элементы совпадают, то кон- тейнер, содержащий меньшее количество элементов, считается меньше, т. к. любой элемент считается больше отсутствующего элемента.

### Модифицирующие алгоритмы

Эта группа алгоритмов позволяет изменить содержимое либо исходного кон- тейнера, либо в процессе копирования данных в другой контейнер.

#### Копирование элементов. Алгоритм *copy()*

OutputIterator copy (InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg)

Алгоритмы копируют все элементы исходного интервала [sourceBeg, sourceEnd) в приемный интервал, начиная с destBeg, и возвращают позицию, следующую за последним скопированным элементом. Причем контейнеры источника и приемника могут быть различными, более того, в качестве ис- точника может выступать итератор входного потока, а приемником может служить итератор выходного потока. Алгоритм copy осуществляет копирова- ние в прямом направлении, а copy\_backward() начинает копировать с конца, сохраняя исходный порядок следования элементов, выходной интервал необ- ходимо задавать с конца. Перед вызовом алгоритма необходимо убедиться, что приемный интервал имеет достаточный размер, иначе возможна ошибка времени выполнения. Рассмотрим в качестве примера, как содержимое мас- сива скопировать в вектор:

OutputIterator copy\_backward (InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destEnd)

vector<int> vec(8); // Создание вектора из 8 элементов copy(arr, arr+8, vec.begin());

for\_each(vec.begin(), vec.end(), print);

Вывод:

4 7 1 6 8 2 6 4

#### Потоковые итераторы

Во многих алгоритмах в качестве входного интервала может применяться входной поток данных, а для выходного интервала часто можно использовать выходной поток. Для реализации этой идеи были разработаны *потоковые итераторы*, которые также реализованы в виде шаблонных классов. Чтобы иметь возможность использовать потоковые итераторы, достаточно подклю- чить файл включений соответствующего потокового класса.

**Класс *ostream\_iterator***

Объекты этого класса могут использоваться в качестве параметров алгорит- ма, использующего выходные итераторы. Он описан в заголовочном файле iostream. Текст программы вывода содержимого контейнера на консоль при помощи алгоритма copy() приведен в листинге 4.8.

**Листинг 4.8**

#include <iostream> #include <algorithm> #include <vector> using namespace std; void main()

{

int arr[] = { 4,7,1,6,8,2,6,4,9,3 };

vector<int> v(arr, arr+10); ostream\_iterator<int> out\_it(cout," "); copy(v.begin(),v.end(),out\_it);

cout << endl;

}

В строке:

ostream\_iterator<int> out\_it(cout," ");

мы определили итератор out\_it для чтения значений типа int. Конструктор, используемый для создания итератора, принимает два параметра, первый из которых — поток вывода cout, а второй — строка-разделитель данных, в на- шем случае это символ пробела.

Аналогично можно вывести данные в файл, если определить поток вывода в файл. Допишем несколько строк в текст программы и добавим файл вклю- чений fstream:

ofstream outfile("out\_iter.dat"); ostream\_iterator<int> out\_file\_it(outfile," "); copy(v.begin(),v.end(),out\_file\_it); outfile.close();

Нетрудно убедиться, что параллельно выводу на консоль в текущей папке появился текстовый файл out\_iter.dat.

**Класс *istream\_iterator***

Если алгоритм использует входные итераторы, можно использовать входной поток. Это, конечно, не самая удачная идея, но попробуем ввести данные чи-

слового вектора с консоли, применяя istream\_iterator. Текст программы, использующей ввод с консоли при помощи алгоритма copy, приведен в лис- тинге 4.9.

**Листинг 4.9**

#include <iostream> #include <algorithm> #include <vector> using namespace std; void main()

{

istream\_iterator<int> in\_it(cin), end\_of\_stream; vector<int> v;

copy(in\_it, end\_of\_stream, back\_inserter(v)); ostream\_iterator<int> out\_it(cout," "); copy(v.begin(),v.end(),out\_it);

cout << endl;

}

Для входного потока необходимо определять два итератора: первый исполь- зуется как текущий и создается конструктором, где в качестве параметра за- дана потоковая переменная in\_it(cin), а второй итератор служит признаком конца потока, он описывается без параметров и создается конструктором по умолчанию end\_of\_stream.

Признак конца потока ввода при вводе с консоли задается клавишами

<Ctrl>+<Z>.



В связи с тем, что поток ввода буферизован, нажимать на <Ctrl>+<Z> придется неоднократно.

В следующем листинге (листинг 4.10) приведен текст программы ввода дан- ных из файла (файл находится в текущей папке).

**Листинг 4.10**

#include <fstream> #include <iostream> #include <algorithm> #include <vector> using namespace std;

void main()

{ ifstream infile("out\_iter.dat"); if (!infile.fail())

{

istream\_iterator<int> file\_it(infile), end\_of\_stream; vector<int> v;

copy(file\_it, end\_of\_stream, back\_inserter(v)); copy(v.begin(), v.end(), ostream\_iterator<int>(cout," ")); cout << endl;

}

}

Мы использовали в качестве выходного итератора итератор вставки back\_inserter(v), что позволяет заполнять вектор, не заботясь о наличии сво- бодного места. Фактически этот итератор вызовет метод push\_back() для вы- ходного контейнера.

Для вывода данных на консоль используем в качестве выходного итератора ostream\_iterator<int>(cout," ") — итератор выходного потока на консоль. Здесь итератор приводится в виде неименованного объекта, который будет создан при обращении к алгоритму и уничтожен при выходе из него.

**Преобразование элементов. Алгоритм *transform()***

OutputIterator transform (InputIterator sourceBeg, InputIterator sourceEnd, OutputIterator destBeg, UnaryPredicate op)

Алгоритм вызывает унарный предикат op(elem) для каждого элемента исход- ного интервала и записывает каждый результат op() в приемный интервал. Позиции sourceBeg и destBeg могут быть идентичны. Возвращает позицию, следующую за последним преобразованным элементом.

Рассмотрим простой пример: читаем набор целых чисел из файла и помещаем данные в контейнер vector с обратным знаком, после чего выводим содержи- мое контейнера на консоль (листинг 4.11).

|  |
| --- |
| **Листинг 4.11** |

#include <fstream> #include <iostream> #include <algorithm> #include <vector> #include <functional> using namespace std; const int N = 100;

void main()

{

ifstream infile("out\_iter.dat");

if (!infile.fail()) // Проверка открытия файла

{

istream\_iterator<int> f\_it(infile), end; vector<int> v;

v.reserve(N); // Задаем размер вектора

// Преобразование данных при чтении файла transform(f\_it, end, back\_inserter(v), negate<int>());

// Вывод вектора на консоль copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

}

else cout << "File not found\n";

}

Вывод:

-4 -7 -1 -6 -8 -2 -6 -4 -9 -3

##### Комбинирование элементов двух интервалов

OutputIterator transform (InputIterator sourceBeg1,

InputIterator sourceEnd1, InputIterator sourceBeg2, OutputIterator destBeg, BinaryPredicate op)

Алгоритм вызывает бинарный предикат op(elem1,elem2) для каждой пары соответствующих элементов двух интервалов [sourceBeg1, sourceEnd1) и [sourceBeg2, ...) и записывает каждый результат в приемный интервал destBeg. Позиции sourceBeg1, sourceBeg2 и destBeg могут быть идентичны. Возвращает позицию за последним преобразованным элементом. В качестве примера приведем текст программы возведения в квадрат всех элементов вектора (листинг 4.12).

|  |
| --- |
| **Листинг 4.12. Программа возведения в квадрат всех элементов вектора** |

#include <iostream> #include <vector> #include <algorithm> #include <functional> using namespace std; void main()

{

int arr[] = { 4,7,1,6,8,2,6,4,9,3 };

vector<int> v(arr, arr+10); copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

transform(v.begin(), v.end(), v.begin(), v.begin(), multiplies<int>());

copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

}

Вывод:

4 7 1 6 8 2 6 4 9 3

16 49 1 36 64 4 36 16 81 9

**Обмен интервалов. Алгоритм *swap\_ranges()***

Iterator swap\_ranges(Iterator beg1, Iterator end1, Iterator beg2)

Алгоритм меняет местами элементы интервала [beg1,end1) с соответствую- щими элементами интервала [beg2,...). Количество элементов определяется размером первого интервала. Возвращает позицию, следующую за последним переставленным элементом во втором интервале.

С помощью этого алгоритма можно, например, поменять в контейнере пре- дыдущего примера 5 первых значений с 5-ю последующими:

swap\_ranges(v.begin(),v.begin()+5,v.begin()+5);

#### Заполнение интервала повторяющимися значениями. Алгоритм *fill()*

void fill (Iterator beg, Iterator end, const T& newValue) void fill\_n (Iterator beg, Size n, const T& newValue)

Алгоритм fill() присваивает значение newValue каждому элементу интервала

[beg,end), а fill\_n() — n элементам, начиная с beg (листинг 4.13).

|  |
| --- |
| **Листинг 4.13** |

#include <iostream> #include <vector> #include <algorithm> using namespace std; void main()

{

vector<int> v(10);

fill(v.begin(),v.begin()+5,1);

fill\_n(v.begin()+5,5,2); copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

}

Вывод:

1 1 1 1 1 2 2 2 2 2

#### Присваивание сгенерированных значений. Алгоритм *generate()*

void generate (Iterator beg, Iterator end, Func op) void generate\_n (Iterator beg, Size n, Func op)

Алгоритм generate() присваивает значение, сгенерированное вызовом op(\*it), каждому элементу интервала [beg,end), а generate\_n — n элемен- там, начиная с beg.

Так, если в предыдущем примере дописать строку:

generate\_n(v.begin(), 10, rand);

получим 10 случайных чисел.

41 18467 6334 26500 19169 15724 11478 29358 26962 24464

Необходимо только добавить файл включений cstdlib, где описана функция генерации случайных чисел rand().

#### Замена элементов внутри интервала. Алгоритм *replace()*

void replace (Iterator beg, Iterator end, const T& oldValue, const T& newValue)

void replace\_if (Iterator beg, Iterator end, UnaryPredicate op, const T& newValue)

Алгоритм replace() заменяет все элементы интервала [beg,end), равные oldValue, на newValue, а replace\_if() производит замену тех элементов, для которых унарный предикат op(elem) возвращает значение true.

Для примера, заменим в предыдущей задаче все 1 на –1.

replace(v.begin(),v.end(),1, -1);

Исходный вектор:

1 1 1 1 1 2 2 2 2 2

После replace():

-1 -1 -1 -1 -1 2 2 2 2 2

#### Замена при копировании

Iterator replace\_copy (Iterator sourceBeg, Iterator sourceEnd, Iterator destBeg, const T& oldValue, const T& newValue)

Iterator replace\_copy\_if (Iterator sourceBeg, Iterator sourceEnd, Iterator destBeg, UnaryPredicate op, const T& newValue)

Алгоритм replace\_copy() заменяет в процессе копирования все элементы ин- тервала [beg,end), равные oldValue, на newValue в приемный интервал, начи- ная с destBeg, а replace\_copy\_if() производит замену тех элементов, для ко- торых унарный предикат op(elem) возвращает значение true. Оба алгоритма возвращают позицию, следующую за последним скопированным элементом в приемном интервале.

Например, в предыдущем примере выведем на консоль вектор, заменив –1 на 1.

replace\_copy (v.begin(),v.end(),ostream\_iterator<int>(cout," "),-1,11);

Исходный вектор:

-1 -1 -1 -1 -1 2 2 2 2 2

После replace\_copy():

11 11 11 11 11 2 2 2 2 2

#### Удаление элементов в интервале. Алгоритм *remove()*

Iterator remove (Iterator beg, Iterator end, const T& Value) Iterator remove\_if (Iterator beg, Iterator end, UnaryPredicate op)

Алгоритм remove() удаляет в интервале [beg,end) все элементы, равные Value, remove\_if() удаляет те элементы, для которых унарный предикат op(elem) возвращает значение true. Удаленные элементы заменяются сле- дующими по порядку. Оба алгоритма возвращают позицию, следующую за последним удаленным элементом.

Например, если удалить из контейнера предыдущего примера все элементы со значением меньше 5, то элементы вектора сдвигаются, заполняя освобо- дившиеся позиции старшими значениями (текст программы — листинг 4.14).

|  |
| --- |
| **Листинг 4.14** |

#include <iostream> #include <vector>

#include <algorithm> #include <functional> using namespace std;

int op (int k) { static int n; return ++n;

}

void main()

{

}

Вывод:

vector<int> v(10); transform(v.begin(),v.end(),v.begin(),op); copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

remove\_if(v.begin(), v.end(), bind2nd(less<int>(),5)); copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

1 2 3 4 5 6 7 8 9 10

5 6 7 8 9 10 7 8 9 10

Обратите внимание, как созданный нами функциональный объект позволил задать числовой вектор в виде арифметической последовательности. Дейст- вительно, здесь переменная задана как static int n, что означает ее инициа- лизацию значением, равным 0, при загрузке функции, и сохранение ее значе- ния при выходе из функции. Таким образом, при каждом обращении к функ- ции переменная увеличивается на 1.

#### Удаление элементов при копировании

Iterator remove\_copy (Iterator sourceBeg, Iterator sourceEnd,

Iterator destBeg, const T& Value)

Iterator remove\_copy\_if (Iterator beg, Iterator end, Iterator destBeg, UnaryPredicate op)

Алгоритм remove\_copy() удаляет в процессе копирования все элементы ин- тервала [sourceBeg, sourceEnd), равные Value, в приемный интервал, начи- нающийся с destBeg. remove\_copy\_if() удаляет те элементы, для которых унарный предикат op(elem) возвращает значение true. Оба алгоритма воз- вращают позицию, следующую за последним скопированным элементом в приемном интервале.

Выведем на консоль преобразованный контейнер с удалением числа 7.

remove\_copy(v.begin(),v.end(),ostream\_iterator<int>(cout," "),7);

Вывод:

5 6 8 9 10 8 9 10

#### Удаление смежных дубликатов. Алгоритм *unique()*

Iterator unique (Iterator beg, Iterator end)

Iterator unique (Iterator beg, Iterator end, BinaryPredicate op)

Обе формы "сворачивают" серии одинаковых элементов в интервале [beg,end), оставляя начальный элемент. Вторая форма удаляет последующие элементы, для которых бинарный предикат op(elem) возвращает значение true.

Алгоритмы заменяют удаленные элементы следующими, которые не были удалены. Возвращают новый логический конец модифицированного интер- вала. Пример программы удаления дубликатов приведен в листинге 4.15.

|  |
| --- |
| **Листинг 4.15. Программа удаления дубликатов** |

#include <iostream> #include <vector> #include <algorithm> #include <functional> using namespace std; void main()

{

int arr[] = { 1,1,1,2,2,2,2,4,5,5 };

vector<int> v(arr, arr+10); copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

vector<int>::iterator end = unique(v.begin(),v.end()); copy(v.begin(), end, ostream\_iterator<int>(cout," ")); cout << endl;

}

Вывод:

1 1 1 2 2 2 2 4 5 5

1 2 4 5

Для корректного вывода уникального набора данных нужно использовать новый логический конец этого набора, который возвращает алгоритм:

vector<int>::iterator end = unique(v.begin(),v.end());

#### Удаление дубликатов при копировании

Iterator unique\_copy (Iterator sourceBeg, Iterator sourceEnd, Iterator destBeg)

Iterator unique\_copy\_if (Iterator sourceBeg, Iterator sourceEnd,

Iterator destBeg, BinaryPredicate op)

Обе формы копируют в приемный интервал, начинающийся с destBeg, все элементы интервала [sourceBeg, sourceEnd), за исключением смежных дуб- ликатов, unique\_copy\_if() удаляет те элементы, для которых бинарный пре- дикат op(elem) возвращает значение true. Оба алгоритма возвращают пози- цию, следующую за последним скопированным элементом в приемном ин- тервале.

Например, если в предыдущем примере нужно вывести уникальный набор на консоль, это можно сделать так:

unique\_copy (v.begin(), v.end(), ostream\_iterator<int>(cout," "));

#### Перестановка элементов в обратном порядке. Алгоритм *reverse()*

Iterator reverse (Iterator beg, Iterator end)

Iterator reverse\_copy (Iterator sourceBeg, Iterator sourceEnd, Iterator destBeg)

Алгоритм reverse() переставляет элементы интервала [beg, end) в обратном порядке, reverse\_copy() — в приемный интервал при копировании. Оба алго- ритма возвращают позицию, следующую за последним скопированным эле- ментом.

В предыдущем примере допишем строку: reverse\_copy(v.begin(), end, ostream\_iterator<int>(cout," ")); и убедимся, что все работает правильно.

#### Циклический сдвиг элементов. Алгоритм *rotate()*

void rotate (Iterator beg, Iterator newBeg, Iterator end) Iterator rotate\_copy (Iterator sourceBeg, Iterator newBeg,

Iterator sourceEnd, Iterator destBeg)

Производит циклический сдвиг элементов интервала [beg, end). После чего

\*newBeg становится новым первым элементом. Алгоритм reverse\_copy() ко- пирует элементы интервала [sourceBeg, sourceEnd) в приемный интервал, начинающийся с destBeg, где \*newBeg является новым первым элементом.

Добавим к нашему примеру еще одну строку.

rotate\_copy(v.begin(),v.begin()+3,end,ostream\_iterator<int>(cout," "));

Несложно убедиться, что четвертый элемент вектора становится первым, а последующие элементы получены путем циклической перестановки.

**Перестановка элементов. Алгоритм *permutation()***

bool next\_permutation (Iterator beg, Iterator end) bool prev\_permutation (Iterator beg, Iterator end)

Алгоритмы изменяют порядок следования элементов в интервале [beg, end) в соответствии со следующей перестановкой next\_permutation() или преды- дущей prev\_permutation(). Оба алгоритма возвращают false, если элементы упорядочены по возрастанию для next\_permutation() или по убыванию для prev\_permutation().

В листинге 4.16 приведен текст программы, в которой в качестве примера приведена перестановка трех букв.

|  |
| --- |
| **Листинг 4.16** |

#include <iostream> #include <vector> #include <algorithm> using namespace std;

void PRINT (vector<char>& v)

{

copy(v.begin(),v.end(),ostream\_iterator<char>(cout," ")); cout << endl;

*}*

void main()

{

char arr[] = { 'b','c','a' }; vector<char> v(arr, arr+3); vector<char> q(v);

cout << "v: "; PRINT(v); cout << "next\n";

while (next\_permutation(v.begin(), v.end())) PRINT(v); cout << "afterward: "; PRINT(v);

cout << "new: "; PRINT(q); cout << "prev\n";

while (prev\_permutation(q.begin(), q.end())) PRINT(q); cout << "afterward: "; PRINT(q);

}

Мы, для простоты, создали отдельную функцию вывода вектора на консоль. Вывод:

v: b c a next

c a b c b a

afterward: a b c new: b c a

prev b a c a c b a b c

afterward: c b a

Здесь нужно обратить внимание на то, что в первом цикле все закончилось исходным порядком afterward: a b c, поскольку это следующая перестанов- ка после c b a. Именно для этой перестановки алгоритм вернул false. После чего можно запускать второй цикл, который заканчивается обратным упоря- дочением.

Таким образом можно сортировать содержимое контейнера, но это будет не- эффективно.

**Алгоритм *random*\_*shuffle()***

void random\_shuffle (Iterator beg, Iterator end)

void random\_shuffle (Iterator beg, Iterator end, RandomFunc op)

Алгоритмы переставляют элементы в интервале [beg, end) в случайном по- рядке. Первая форма использует генератор случайных чисел с равномерным распределением, вторая использует операцию op(max), где max — размер ин- тервала.

Рассмотрим в качестве примера перестановку массива целых чисел, где мы используем алгоритм генерации случайных чисел, предложенный Страустру- пом. Текст данной программы приведен в листинге 4.17.

|  |
| --- |
| **Листинг 4.17** |

#include <stdlib.h> #include <iostream> #include <vector> #include <algorithm> using namespace std;

void PRINT (vector<int>& v)

{ copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl; }

class MyRandom

{ public: int operator() (int max)

{ double tmp = double(rand())/double(RAND\_MAX); return int(tmp\*max);

}// Здесь RAND\_MAX = 0x7fff;

};

void main()

{

vector<int> Vi(10);

for (int i = 0; i < 10; i++) Vi[i] = i; PRINT(Vi);

MyRandom rd; random\_shuffle(Vi.begin(),Vi.end(),rd); PRINT(Vi);

}

Вывод:

0 1 2 3 4 5 6 7 8 9

3 2 7 4 1 0 5 9 6 8

**Перемещение элементов в начало. Алгоритм *partition()***

Iterator partition (Iterator beg, Iterator end, UnaryPredicate op) Iterator stable\_partition (Iterator beg, Iterator end, UnaryPredicate op)

Оба алгоритма перемещают в начало интервала [beg, end) все элементы, для которых унарный предикат op(elem) возвращает true. Возвращают первую позицию, для которой op(elem) возвращает false. Алгоритм stable\_partition() сохраняет относительный порядок следования элементов.

Пример программы, демонстрирующий способ перемещения в начало диапа- зона нечетных чисел, приведен в листинге 4.18.

**Листинг 4.18**

#include <iostream> #include <vector> #include <algorithm> using namespace std;

void PRINT (vector<int>& v)

{

copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

}

bool op(int elem) { return (elem & 1) > 0; } void main()

{

vector<int> v(10);

for (int i = 0; i < 10; i++) v[i] = i; PRINT(v);

vector<int> q(v); partition(v.begin(),v.end(),op);

cout << "partition: "; PRINT(v); stable\_partition(q.begin(), q.end(), op); cout << "stable\_partition: "; PRINT(q);

}

Вывод:

0 1 2 3 4 5 6 7 8 9

partition(): 9 1 7 3 5 4 6 2 8 0

stable\_partition(): 1 3 5 7 9 0 2 4 6 8

#### Сортировка всех элементов. Алгоритмы сортировки *sort()*

void sort (Iterator beg, Iterator end)

void sort (Iterator beg, Iterator end, BinaryPredicate op) void stable\_sort (Iterator beg, Iterator end)

void stable\_sort (Iterator beg, Iterator end, BinaryPredicate op)

Алгоритмы сортируют все элементы интервала [beg, end) оператором < или используют в качестве критерия сортировки бинарный предикат op(elem1, elem2). Отличие алгоритма sort() от stable\_sort() заключается в том, что stable\_sort() сохраняет относительный порядок следования рав- ных элементов.



Алгоритмы сортировки не могут использоваться со списками, поскольку списки не поддерживают итераторы произвольного доступа. Для списков реализован метод sort().

Отсортируем массивы целых чисел предыдущего примера:

sort(v.begin(),v.end());

cout << "sort v on increase: "; PRINT(v);

sort(q.begin(), q.end(), greater<int>()); cout << "sort q on decrease: "; PRINT(q);

Вывод:

sort v on increase: 0 1 2 3 4 5 6 7 8 9

sort q on decrease: 9 8 7 6 5 4 3 2 1 0

#### Простая частичная сортировка

void partial\_sort (Iterator beg, Iterator SortEnd, Iterator end) void partial\_sort (Iterator beg, Iterator SortEnd, Iterator end,

BinaryPredicate op)

Алгоритмы сортируют все элементы интервала [beg, end) оператором < или используют в качестве критерия сортировки бинарный предикат op(elem1, elem2) так, чтобы интервал [beg, SortEnd) содержал упорядочен- ные данные. В отличие от алгоритма sort(), они прекращают работу, как только интервал [beg, SortEnd) будет упорядочен.

В предыдущей задаче изменим одну строку: partial\_sort(q.begin(),q.begin()+5, q.end(), greater<int>()); Вывод:

9 8 7 6 5 0 1 2 3 4

Действительно, отсортированы только первые пять значений, остальные со- хранили исходное упорядочение.

#### Частичная сортировка с копированием

void partial\_sort\_copy (Iterator sourceBeg, Iterator sourceEnd,

Iterator destBeg, Iterator destEnd) void partial\_sort\_copy (Iterator sourceBeg, Iterator sourceEnd,

Iterator destBeg, Iterator destEnd, BinaryPredicate op)

Алгоритмы сортируют элементы из интервала [sourceBeg, sourceEnd) в при- емный интервал [destBeg, destEnd). Количество упорядоченных и скопиро- ванных элементов равно минимальному количеству элементов в исходном и приемном интервалах. Алгоритмы возвращают позицию, следующую за по- следним скопированным элементом в приемном интервале.

Так, в предыдущем примере мы заменим алгоритм partial\_sort() на

partial\_sort\_copy() и выведем данные на консоль.

partial\_sort\_copy(q.begin(),q.end(),v.begin(),v.begin()+5, greater<int>());

Вывод:

9 8 7 6 5 5 6 7 8 9

В приемный интервал помещены первые 5 элементов, отсортированных по убыванию, последние 5 элементов приемного интервала остались без изме- нений.

#### Разбиение по n-му элементу

void nth\_element (Iterator beg, Iterator nth, Iterator end) void nth\_element (Iterator beg, Iterator nth, Iterator end,

BinaryPredicate op)

Алгоритмы сортируют элементы интервала [beg, end) так, чтобы элемент

\*nth занимал правильное в процессе сортировки место, т. е. все элементы слева меньше его, а последующие — больше или равны.

Например, если в предыдущей задаче мы хотели бы разделить интервал на значения больше и меньше 6, тогда с помощью алгоритма find() найдем по- зицию числа 6 и воспользуемся алгоритмом nth\_element().

nth\_element(v.begin(),find(v.begin(),v.end(),6),v.end());

Вывод:

5 5 6 6 7 7 8 8 9 9

### Алгоритмы упорядоченных интервалов

Алгоритмы упорядоченных интервалов требуют, чтобы элементы исходных контейнеров были упорядочены. Они гораздо эффективнее стандартных ана- логов, поэтому, если исходное упорядочение элементов контейнеров не со- ставляет большой проблемы, стараются использовать именно эту группу ал- горитмов. Однако для неупорядоченных контейнеров эти алгоритмы могут привести к непредсказуемым последствиям.

**Проверка присутствия элемента. Алгоритм *binary\_search()***

bool binary\_search (Iterator beg, Iterator end, const T& val) bool binary\_search (Iterator beg, Iterator end, const T& val,

BinaryPredicate op)

Алгоритм проверяет, содержит ли интервал [beg, end) элемент со значени- ем val. Бинарный предикат определяет критерий сортировки.

В листинге 4.19 приведен текст программы проверки содержания в числовом векторе значения, равного 5.

**Листинг 4.19**

#include <iostream> #include <vector> #include <algorithm> using namespace std; void main()

{

vector<int> v(10);

for (int i = 0; i < 10; i++) v[i] = i; if(binary\_search(v.begin(),v.end(),5))cout << "5 is in vector\n";

}

#### Проверка присутствия подинтервала. Алгоритм *includes()*

bool includes (Iterator beg, Iterator end, Iterator searchBeg, Iterator searchEnd)

bool includes (Iterator beg, Iterator end, Iterator searchBeg, Iterator searchEnd, BinaryPredicate op)

Алгоритм проверяет, содержится ли подинтервал [searchBeg, searchEnd) в интервале [beg, end). Бинарный предикат определяет критерий сортировки.

Например, проверим, содержится ли в числовом векторе предыдущего при- мера вектор со значениями {2, 3, 4}. Добавим несколько строк к программе.

int arr[] = { 2, 3, 4}; vector<int> q(arr, arr+3);

if(includes(v.begin(), v.end(), q.begin(), q.end())) cout << "q is in vector v\n";

#### Поиск первой или последней возможной позиции. Алгоритм *lover\_bound()*

Iterator lover\_bound (Iterator beg, Iterator end, const T& val) Iterator lover\_bound (Iterator beg, Iterator end, const T& val, BinaryPredicate op)

Алгоритм возвращает позицию первого элемента со значением большим либо равным val, куда оно может быть вставлено без нарушения упорядочения. Если позиция не найдена, алгоритм возвращает end. Бинарный предикат оп- ределяет критерий сортировки.

**Алгоритм *upper\_bound()***

Iterator upper\_bound (Iterator beg, Iterator end, const T& val) Iterator upper\_bound (Iterator beg, Iterator end, const T& val, BinaryPredicate op)

Алгоритм возвращает позицию последнего элемента со значением большим val, куда оно может быть вставлено без нарушения упорядочения. Если пози- ция не найдена, алгоритм возвращает end. Бинарный предикат определяет критерий сортировки.

#### Поиск первой и последней возможных позиций. Алгоритм *equal\_range()*

pair<Iterator,Iterator> equal\_range (Iterator beg, Iterator end,

const T& val) pair<Iterator,Iterator> equal\_range (Iterator beg, Iterator end,

const T& val, BinaryPredicate op)

Алгоритм возвращает парой итераторов начало и конец интервала значений, равных val, куда может быть вставлено это значение без нарушения упоря- дочения. (Шаблонная структура pair приведена в листинге 4.7.) Если пози- ция не найдена, оба итератора равны end. Бинарный предикат определяет критерий сортировки.

Текст программы поиска для алгоритмов упорядоченных диапазонов приве- ден в листинге 4.20.

**Листинг 4.20**

#include <iostream> #include <vector> #include <algorithm> using namespace std; void main()

{

int arr[] = { 2, 3, 4, 6, 6, 6, 6, 9, 9, 9 };

vector<int> v(arr, arr+10); vector<int>::iterator it;

it = lower\_bound(v.begin(), v.end(),6);

cout << "Index >= 6: " << it - v.begin() << endl; it = upper\_bound(v.begin(), v.end(),6);

cout << "Index > 6: " << it - v.begin() << endl;

// Теперь то же самое, но в алгоритме equal\_range pair<vector<int>::iterator, vector<int>::iterator> pair\_it;

pair\_it = equal\_range(v.begin(), v.end(),6);

cout << "Index >= 6: " << pair\_it.first - v.begin() << endl; cout << "Index > 6: " << pair\_it.second - v.begin() << endl;

}

Вывод:

Index >= 6: 3

Index > 6: 7

#### Слияние интервалов. Алгоритм *merge()*

OutIterator merge (Iterator beg1, Iterator end1,

Iterator beg2, Iterator end2, OutIterator destBeg) OutIterator merge (Iterator beg1, Iterator end1, Iterator beg2,

Iterator end2, OutIterator destBeg, BinaryPredicate op)

Алгоритм комбинирует элементы упорядоченных интервалов таким образом, что приемный интервал содержит все элементы обеих исходных интервалов с сохранением упорядочения. Возвращает позицию, следующую за последним скопированным элементом. Бинарный предикат определяет порядок сорти- ровки. Приемный интервал не должен перекрывать исходный.

В листинге 4.21 приведен текст программы, в которой продемонстрирован пример слияния двух числовых массивов.

**Листинг 4.21**

#include <iostream> #include <vector> #include <algorithm> using namespace std; void main()

{

int arr1[] = { 1, 3, 5, 7, 9 };

int arr2[] = { 2, 4, 6, 8, 10, 12 };

vector<int> v1(arr1, arr1 + 5); vector<int> v2(arr2, arr2 + 6); vector<int> v(20); vector<int>::iterator end;

end=merge(v1.begin(), v1.end(), v2.begin(), v2.end(), v.begin()); copy(v.begin(), end, ostream\_iterator<int>(cout," "));

cout << endl;

}

Вывод:

1 2 3 4 5 6 7 8 9 10 12

#### Объединение двух упорядоченных множеств элементов. Алгоритм *set\_union()*

OutIterator set\_union (Iterator beg1, Iterator end1, Iterator beg2,

Iterator end2, OutIterator destBeg) OutIterator set\_union (Iterator beg1, Iterator end1, Iterator beg2,

Iterator end2, OutIterator destBeg,BinaryPredicate op)

Алгоритм комбинирует элементы упорядоченных интервалов таким образом, что приемный интервал содержит все элементы, присутствующие в первом или втором интервалах, с сохранением упорядочения. При наличии повто- ряющихся значений в приемный интервал копируется не суммарное их коли- чество, а максимальное. Возвращает позицию, следующую за последним ско- пированным элементом. Бинарный предикат определяет порядок сортировки. Приемный интервал не должен перекрывать исходный.

Так, если в предыдущем примере добавить упорядоченный массив, то мы по- лучим:

int m[] = { 1, 1, 4, 5, 5, 5, 9, 10 };

vector<int> v3(m, m + 8);

end=set\_union(v1.begin(), v1.end(), v3.begin(), v3.end(), v.begin()); copy(v.begin(), end, ostream\_iterator<int>(cout," "));

cout << endl;

Вывод:

1 1 3 4 5 5 5 7 9 10

**Пересечение двух упорядоченных множеств элементов. Алгоритм *set\_intersection()***

OutIterator set\_intersection (Iterator beg1, Iterator end1, Iterator beg2,

Iterator end2, OutIterator destBeg) OutIterator set\_intersection (Iterator beg1, Iterator end1, Iterator beg2,

Iterator end2, OutIterator destBeg, BinaryPredicate op)

Алгоритм комбинирует элементы упорядоченных интервалов таким образом, что приемный интервал содержит все элементы, присутствующие сразу в обоих интервалах с сохранением упорядочения. Причем, при наличии повто- ряющихся значений, в приемный интервал копируется не суммарное их ко- личество, а минимальное. Возвращает позицию, следующую за последним

скопированным элементом. Бинарный предикат определяет порядок сорти- ровки. Приемный интервал не должен перекрывать исходный.

Например, если в предыдущем примере заменить алгоритм set\_union() на

set\_intersection(), то мы получим:

1 5 9

**Разность двух упорядоченных множеств элементов. Алгоритм *set\_difference()***

OutIterator set\_difference (Iterator beg1, Iterator end1,

Iterator beg2, Iterator end2, OutIterator destBeg) OutIterator set\_difference (Iterator beg1, Iterator end1,

Iterator beg2, Iterator end2, OutIterator destBeg, BinaryPredicate op)

Алгоритм комбинирует элементы упорядоченных интервалов таким образом, что приемный интервал содержит все элементы, присутствующие в первом, но не входящие во второй интервал, с сохранением упорядочения. Причем, при наличии повторяющихся значений, их количество в приемном интервале равно разности между количеством в первом и втором интервалах. Если ко- личество дубликатов во втором интервале больше, то в приемном интервале они будут отсутствовать. Возвращает позицию за последним скопированным элементом. Бинарный предикат определяет порядок сортировки. Приемный интервал не должен перекрывать исходный.

Например, если в предыдущем примере заменить алгоритм на set\_difference(), то мы получим:

3 7

**Алгоритм *set\_symmetric\_difference()***

OutIterator set\_symmetric\_difference (Iterator beg1, Iterator end1,

Iterator beg2, Iterator end2, OutIterator destBeg) OutIterator set\_symmetric\_difference (Iterator beg1, Iterator end1,

Iterator beg2, Iterator end2, OutIterator destBeg, BinaryPredicate op)

Алгоритм комбинирует элементы упорядоченных интервалов таким образом, что приемный интервал содержит все элементы, присутствующие либо в пер- вом, либо во втором интервалах, но не в обоих интервалах. Причем, при на- личии повторяющихся значений, их количество в приемном интервале равно разности между количеством в первом и втором интервалах. Возвращает по- зицию, следующую за последним скопированным элементом. Бинарный пре- дикат определяет порядок сортировки. Приемный интервал не должен пере- крывать исходный.

Например, если заменить алгоритм на set\_symmetric\_difference() в преды- дущем примере, то мы получим:

1 3 4 5 5 7 10

**Слияние смежных упорядоченных интервалов. Алгоритм *inplace\_merge()***

void inplace\_merge (Iterator beg1, Iterator end1beg2, Iterator end2) void inplace\_merge (Iterator beg1, Iterator end1beg2, Iterator2 end2,

BinaryPredicate op)

Алгоритм выполняет слияние двух смежных интервалов [beg1..end1beg2.. end2) так, чтобы полный интервал [beg1, end2) содержал упорядоченную совокупность элементов обоих интервалов. Бинарный предикат определяет порядок сортировки.

Например, создадим вектор из числового массива и воспользуемся этим ал- горитмом:

int mn[] = { 2, 4, 6, 8, 10, 1, 3, 5, 7, 9 };

vector<int> v4(mn, mn + 10); inplace\_merge(v4.begin(), v4.begin() + 5, v4.end());

copy(v4.begin(), v4.end(), ostream\_iterator<int>(cout," ")); cout << endl;

Вывод:

1 2 3 4 5 6 7 8 9 10

### Численные алгоритмы

Алгоритмы, предназначенные для обработки численных данных, представле- ны в заголовочном файле numeric.

#### Суммирование элементов интервала

T accumulate (Iterator beg, Iterator end, T initValue)

Для всех элементов интервала выполняется команда:

initValue = initValue + elem;

Для всех элементов интервала вычисляется сумма + начальное значение

initValue. Возвращается накопленное значение суммы.

#### Вычисления с накоплением результата

T accumulate (Iterator beg, Iterator end, T initValue, BinaryFunc op)

Для всех элементов интервала выполняется команда:

initValue = op(initValue, elem);

Возвращается накопленное значение initValue.

Здесь есть возможность выполнить более сложные вычисления.

Например, в листинге 4.22 представлена программа вычисления суммы и произведения элементов числового вектора.

|  |
| --- |
| **Листинг 4.22. Вычисление суммы и произведения элементов числового вектора** |

#include <iostream> #include <vector> #include <numeric> #include <functional> using namespace std; void main()

{

int arr[] = { 1, 3, 5, 7, 9 };

vector<int> v(arr, arr + 5); int sum, product;

sum = accumulate(v.begin(), v.end(), 0);

product = accumulate(v.begin(), v.end(), 1, multiplies<int>()); cout << "Sum = " << sum << endl;

cout << "Product = " << product << endl;

}

Вывод:

Sum = 25

Product = 945

#### Вычисление скалярного произведения

T inner\_product (Iterator1 beg1, Iterator1 end1, Iterator2 beg2, T initValue)

Алгоритм вычисляет скалярное произведение двух векторов из интервала

[beg1, end1) и [beg1...). Для всех элементов выполняется команда:

initValue = initValue + elem1\*elem2;

Возвращается накопленное значение initValue.

#### Вычисления с двумя интервалами

T inner\_product (Iterator beg1, Iterator end1, Iterator beg2, T *initValue*, BinaryFunc op1, BinaryFunc op2)

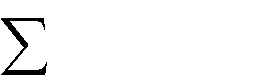
Для всех элементов из интервала [beg1, end1) и [beg1...) выполняется команда:

initValue = op1(initValue, op2(elem1,elem2));

Возвращается накопленное значение initValue.

Для примера вычислим скалярное произведение вектора v и вектора v1 (см. задачу в листинге 4.22), а также вычислим сумму квадратов разностей коор- динат двух векторов.

*d*



(*vi v*1*i* )2 .

*i*

Для этого нам придется описать функциональный объект op(): int op(int elem1, int elem2)

{

int tmp = elem1 - elem2; return tmp\*tmp;

}

и дописать необходимый код.

int arr1[] = { 2, 4, 6, 8, 10 };

vector<int> v1(arr1, arr1 + 5); int scalar\_product, d;

scalar\_product = inner\_product(v.begin(), v.end(), v1.begin(), 0); cout << "scalar\_product = " << scalar\_product << endl;

d = inner\_product(v.begin(), v.end(), v1.begin(), 0, plus<int>(), op); cout << "d = " << d << endl;

Вывод:

scalar\_product = 190

d = 5

#### Накопительное суммирование элементов интервала

Iterator partial\_sum(Iterator beg, Iterator end, Iterator destBeg) Iterator partial\_sum(Iterator beg, Iterator end, Iterator destBeg,

BinaryFunc op)

Алгоритм вычисляет и направляет в приемный интервал для всех элементов следующие выражения:

a1, a1+a2, a1+a2+a3, ...

a1, a1 op a2, a1 op a2 op a3, ...

Возвращает позицию, следующую за последним записанным элементом в приемном интервале. Исходный и приемный интервалы могут совпадать.

С помощью этого алгоритма можно, например, создать арифметическую и геометрическую прогрессию.

Пример текста программы, демонстрирующей работу с численными алго- ритмами, приведен в листинге 4.23.

**Листинг 4.23**

#include <iostream> #include <vector> #include <numeric> #include <functional> using namespace std; void main()

{

}

Вывод:

vector<int> v(10,2), v1(v); partial\_sum(v.begin(), v.end(), v.begin());

copy(v.begin(),v.end(),ostream\_iterator<int>(cout," ")); cout << endl;

partial\_sum(v1.begin(),v1.end(),v1.begin(),multiplies<int>()); copy(v1.begin(),v1.end(),ostream\_iterator<int>(cout," ")); cout << endl;

2 4 6 8 10 12 14 16 18 20

2 4 8 16 32 64 128 256 512 1024

#### Преобразование абсолютных значений в относительные

Iterator adjacent\_difference (Iterator beg, Iterator end,

Iterator destBeg)

Iterator adjacent\_difference (Iterator beg, Iterator end,

Iterator destBeg, BinaryFunc op)

Алгоритм вычисляет и направляет в приемный интервал для всех элементов следующие выражения:

a1, a2-a1, a3-a2, ...

a1, a2 op a1, a3 op a2, ...

Возвращает позицию, следующую за последним записанным элементом в приемном интервале. Исходный и приемный интервалы могут совпадать.

Алгоритм является логическим антиподом алгоритма partial\_sum(). С его помощью можно, например, воссоздать исходное состояние контейнеров предыдущего примера:

adjacent\_difference (v.begin(), v.end(), v.begin()); adjacent\_difference (v1.begin(), v1.end(), v1.begin(), divides<int>());

**Контейнер *Deque***

*Дек* реализован в виде динамического массива и состоит из нескольких не- смежных блоков (рис. 4.2). В отличие от вектора, дек открыт для расширения с обоих концов, что позволяет наращивать его в противоположных направле- ниях. В связи с таким строением, при необходимости перераспределения па- мяти не нужно перемещать содержимое контейнера целиком, достаточно лишь переписать один или несколько блоков. Таким образом, дек работает более эффективно с операциями вставки внутрь контейнера.

**Рис. 4.2.** Структура дека

Чтобы использовать дек, в программе необходимо подключить заголовочный файл:

#include <deque>

### Операции с деками

Интерфейсы дека и вектора мало чем отличаются друг от друга. Большинство их методов совпадают по своим функциональным возможностям. Принципи- альное же отличие заключается в возможности вставки новых элементов как в конец, так и в начало дека. Эти операции выполняются с одинаковой эф- фективностью.

Операции с деками рассмотрим по группам, составив соответствующие таб- лицы: конструкторы, немодифицирующие операции, модифицирующие опе- рации (табл. 4.10—4.12).

***Таблица 4.10.*** *Конструкторы*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| deque<T> d | Создает пустой дек |
| deque<T> d(n) | Создает дек из n элементов |
| deque<T> d2(v1) | Создает копию дека d1 |
| deque<T> d2(n,elem) | Создает дек из n копий elem |
| deque<T> d2(beg,end) | Создает дек из элементов интервала [beg,end) контей- нера deque. В отличие от остальных контейнеров, дек не позволяет использовать другие контейнеры в конструк- торе |

***Таблица 4.11.*** *Немодифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| d.size() | Возвращает размер дека |
| d.empty() | true, если дек пуст |
| d.max\_size() | Возвращает максимальный размер дека |
| d1 == d2 | Проверяет равенство |
| d1 != d2 | Проверяет неравенство |
| d1 < d2 | Проверяет, что d1 < d2 |
| d1 > d2 | Проверяет, что d1 > d2 |
| d1 <= d2 | Проверяет, что d1 <= d2 |
| d1 >= d2 | Проверяет, что d1 >= d2 |
| d[index] | Возвращает элемент с индексом index |
| d.at(index) | Возвращает элемент с индексом index. Генерируется исключе- ние out\_of\_range при выходе за границы интервала |
| d.front() | Возвращает первый элемент |
| d.back() | Возвращает последний элемент |
| d.begin() | Возвращает итератор произвольного доступа первого элемента |
| d.end() | Возвращает итератор произвольного доступа за последним эле- ментом |
| d.rbegin() | Возвращает обратный итератор первого элемента для перебора в обратном направлении |
| d.rend() | Возвращает обратный итератор последнего элемента для пере- бора в обратном направлении |

***Таблица 4.12.*** *Модифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| d1 = d2 | Присваивает d1 все элементы d2 |
| d.assign(n,elem) | Присваивает n копий elem |
| d.assign(beg,end) | Присваивает деку d элементы контейнера deque [beg,end) |
| d1.swap(d2) | Меняет местами содержимое d1 и d2 |
| swap(d1,d2) | То же в форме функции |
| d.insert(pos,elem) | Вставляет в позицию pos элемент, возвращает пози- цию следующего элемента |
| d.insert(pos,n,elem) | Вставляет в позицию pos n элементов elem |
| d.insert(pos,beg,end) | Вставляет элементы интервала [beg,end) в пози- цию pos |
| d.push\_back(elem) | Вставляет элемент elem в конец контейнера |
| d.pop\_back() | Удаляет последний элемент |
| d.push\_front(elem) | Вставляет элемент elem в начало дека |
| d.pop\_front() | Удаляет первый элемент |
| d.erase(pos) | Удаляет элемент в позиции pos и возвращает пози- цию следующего элемента |
| d.erase(beg,end) | Удаляет все элементы из интервала [beg,end) и возвращает позицию следующего элемента |
| d.resize(num) | Приводит контейнер к размеру num |
| d.resize(num,elem) | Приводит контейнер к размеру num, новые элементы создаются как копии elem |
| d.clear() | Удаляет содержимое дека |

В листинге 4.24 приведен текст программы, тестирующей основные методы дека.

|  |
| --- |
| **Листинг 4.24** |

#include <algorithm> #include <deque> #include <iostream> using namespace std;

void main()

{// Конструкторы:

deque<int> d1; deque<int> d2(10); deque<int> d3(10,1); deque<int> d4(d3);

cout << "size d4 = " << d4.size() << endl;

cout << "Max size deque<int> = " << d4.max\_size() << endl; if (d1.empty()) cout << "d1 is empty\n";

if (d3 == d4) cout << "d3 == d4\n"; else cout << "d3 != d4\n";

// Заполнение дека элементами массива: int arr[5] = { -1, 3, 5, 8, 2 };

deque<int> d5;

for (int i = 0; i < 5; i++) d5.push\_back(arr[i]); d1.assign(d5.begin(),d5.end()); // Копирование дека copy(d1.begin(),d1.end(),ostream\_iterator<int>(cout," ")); cout << endl;

// Обращение к элементам дека по индексу:

for (i = 0; i < d2.size();i++) d2[i] = i+1; copy(d2.begin(),d2.end(),ostream\_iterator<int>(cout," ")); cout << endl;

// Конструктор копирует содержимое дека d5: deque<int> deq(d5.begin(), d5.end());

// Обращение к элементам дека через итератор: deque<int>::iterator it;

for (it = deq.begin(); it != deq.end(); it++) cout << \*it << ' '; cout << endl;

}

Вывод:

size d4 = 10

Max size deque<int> = 1073741823 d1 is empty

d3 == d4

-1 3 5 8 2

1 2 3 4 5 6 7 8 9 10

-1 3 5 8 2

При работе с деком, да и со всеми контейнерами, кроме вектора, следует иметь в виду, что нет никаких оснований считать элементы контейнера рас- положенными в непрерывной области памяти. Более того, как правило, кон- тейнер занимает несколько разрозненных блоков памяти, в связи с чем опе-

рация '<' или '>' с итераторами не имеет большого смысла, поэтому при передвижении по контейнеру обычно используют сравнение вида:

it != deq.end().

**Контейнер *List***

По своей внутренней структуре *списки* принципиально отличаются от векто- ров и деков (рис. 4.3). Каждый элемент списка, кроме данных, содержит ссылки на предыдущий и последующий элемент. Такая организация позволя- ет вставлять и удалять данные не только в начало и конец списка, но и в про- извольное место в нем при одной и той же производительности.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

**Рис. 4.3.** Организация списка

Списки не поддерживают итераторы произвольного доступа, в них не опре- делены ни оператор индексирования [], ни метод at(). Для доступа к элемен- ту списка необходимо перебрать все предшествующие элементы. В связи с этим списки не поддерживают работу с некоторыми алгоритмами. В частно- сти, со списком не может быть реализован алгоритм сортировки, однако спи- сок имеет собственные методы сортировки.

Чтобы использовать список в программе, необходимо подключить заголо- вочный файл list.

Операции со списками рассмотрим по группам, составив соответствующие таблицы: конструкторы, немодифицирующие операции, модифицирующие операции, специальные операции (табл. 4.13—4.16).

***Таблица 4.13.*** *Конструкторы*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| list<T> lst | Создает пустой список |
| list<T> lst (n) | Создает список из n элементов |
| list<T> lst1(lst) | Создает копию списка lst |
| list<T> lst (n,elem) | Создает список из n копий elem |
| list<T> lst (beg,end) | Создает список из элементов интервала [beg,end) |

***Таблица 4.14.*** *Немодифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| lst.size() | Возвращает размер списка |
| lst.empty() | Возвращает true, если список пуст |
| lst.max\_size() | Возвращает максимальный размер списка |
| lst1 == lst2 | Проверяет равенство |
| lst1 != lst2 | Проверяет неравенство |
| lst1 < lst2 | Проверяет, что lst1 < lst2 |
| lst1 > lst2 | Проверяет, что lst1 > lst2 |
| lst1 <= lst2 | Проверяет, что lst1 <= lst2 |
| lst1 >= lst2 | Проверяет, что lst1 >= lst2 |
| lst.front() | Возвращает первый элемент |
| lst.back() | Возвращает последний элемент |
| lst.begin() | Возвращает итератор первого элемента |
| lst.end() | Возвращает итератор за последним элементом |
| lst.rbegin() | Возвращает обратный итератор начала контейнера для пере- бора в обратном направлении |
| lst.rend() | Возвращает обратный итератор конца контейнера для пере- бора в обратном направлении |

***Таблица 4.15.*** *Модифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| lst1 = lst2 | Присваивает списку lst1 все элементы lst2 |
| lst.assign(n,elem) | Присваивает n копий elem |
| lst.assign(beg,end) | Присваивает контейнеру list элементы интервала  [beg,end) |
| lst1.swap(lst2) | Меняет местами содержимое lst1 и lst2 |
| swap(lst1,lst2) | То же в форме функции |
| lst.insert(pos,elem) | Вставляет в позицию pos элемент, возвращает пози- цию следующего элемента |
| lst.insert(pos,n,elem) | Вставляет в позицию pos n элементов elem |
| lst.insert(pos,beg,end) | Вставляет элементы интервала [beg,end) с позиции  pos |

***Таблица 4.15*** *(окончание)*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| lst.push\_back(elem) | Вставляет элемент elem в конец списка |
| lst.pop\_back() | Удаляет последний элемент |
| lst.push\_front(elem) | Вставляет элемент elem в начало списка |
| lst.pop\_front() | Удаляет первый элемент |
| lst.remove(val) | Удаляет все элементы со значением val |
| lst.remove\_if(op) | Удаляет все элементы, для которых op(elem) воз- вращает true |
| lst.erase(pos) | Удаляет элемент в позиции pos и возвращает пози- цию следующего элемента |
| lst.erase(beg,end) | Удаляет все элементы из интервала [beg,end) и возвращает позицию следующего элемента |
| lst.resize(num) | Приводит контейнер к размеру num |
| lst.resize(num,elem) | Приводит контейнер к размеру num, новые элементы создаются как копии elem |
| lst.clear() | Удаляет содержимое списка |

Для лучшего понимания технологии работы со списками на рис. 4.4 и 4.5 по- кажем графически, как происходят вставка и удаление элемента внутри списка.



**Рис. 4.4.** Вставка элемента списка



**Рис. 4.5.** Удаление элемента списка

***Таблица 4.16.*** *Специальные операции списка*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| lst.unique() | Удаляет дубликаты |
| lst.unique(op) | Удаляет дубликаты, для которых op возвращает  true |
| lst.splice(pos,lst1) | Перемещает все элементы lst1 в lst перед пози- цией итератора pos |
| lst.splice(pos,lst1,pos1) | Перемещает все элементы lst1, начиная с пози- ции итератора pos1, в lst перед позицией итера- тора pos |
| lst.splice(pos,lst1,  beg1,end1) | Перемещает все элементы интервала [beg1,end1)  списка lst1 в lst перед позицией итератора pos |
| lst.sort() | Сортирует все элементы оператором < |
| lst.sort(op) | Сортирует все элементы по критерию op |
| lst.merge(lst1) | Перемещает все элементы lst1 в lst с сохране- нием сортировки |
| lst.merge(lst1,op) | Перемещает все элементы lst1 в lst с сохране- нием сортировки по критерию op |
| lst.reverse() | Переставляет все элементы в обратном порядке |

В листинге 4.25 приведен текст программы примера использования списка.

**Листинг 4.25**

#include <algorithm> #include <list> #include <iostream> #include <functional> using namespace std; void main()

{

int arr[5] = { -1, 3, 5, 8, 2 };

// Конструкторы

list<int> lst; list<int> lst1(10); list<int> lst2(10,2); list<int> lst3(lst2);

list<int> lst4(arr, arr+5);

// Методы

cout << "Size lst3 = " << lst3.size() << endl;

cout << "Max Size lst3 = " << lst3.max\_size() << endl; if (lst.empty()) cout << "lst is empty\n";

if(lst3 == lst2)cout<<"lst3==lst2\n";else cout<<"lst3 != lst2\n"; lst = lst4; // Присваивание

lst.sort(); // Сортировка по умолчанию

lst.splice(lst.end(), lst4); // Добавим исходный список в конец copy(lst.begin(),lst.end(),ostream\_iterator<int>(cout," ")); cout << endl;

lst.sort(greater<int>()); // Сортируем по убыванию copy(lst.begin(),lst.end(),ostream\_iterator<int>(cout," ")); cout << endl;

lst.unique(); // Удаляем дубликаты copy(lst.begin(),lst.end(),ostream\_iterator<int>(cout," ")); cout << endl;

lst.reverse(); // Обращение массива данных copy(lst.begin(),lst.end(),ostream\_iterator<int>(cout," ")); cout << endl;

}

**Контейнеры *Set, Multiset***

*Множества* и *мультимножества* автоматически сортируют элементы, по умолчанию используя критерий сортировки <. Они отличаются только тем, что мультимножества могут содержать дубликаты, а множества — нет. Что- бы использовать эти контейнеры, необходимо включить в программу заголо- вочный файл:

#include <set> или #include <multiset>

Особенности строения множеств (здесь и далее под множеством будем пони- мать также и мультимножество), а они реализованы в виде сбалансированно- го бинарного дерева, приводит к тому, что на множестве нельзя напрямую менять значение элементов, иначе это нарушит правильный порядок их рас- положения. Следовательно, чтобы изменить значение элемента, необходимо этот элемент сначала удалить из множества, затем вставить его с новым зна- чением.

Практические следствия такой особенности множества, как контейнера, при- водят к тому, что:

* не поддерживается прямое обращение к элементам;
* при косвенном обращении через итераторы значение элемента является константным.

Операции с множествами и мультимножествами рассмотрим по группам, со- ставив соответствующие таблицы: конструкторы, немодифицирующие опе- рации, модифицирующие операции (табл. 4.17—4.19).

***Таблица 4.17.*** *Конструкторы*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| set<T> c | Создает пустое множество |
| set<T> c(op) | Создает пустое множество с критерием сортировки op |
| set<T> c1(c) | Создает копию множества |
| set<T> c(beg,end) | Создает множество из элементов интервала [beg,end). Исходный контейнер может быть любого типа (от масси- ва до входного потока) |
| set<T> c(beg,end,op) | Создает множество с критерием сортировки op из эле- ментов интервала [beg,end) |



Для мультимножества вместо set пишем multiset.

Существует два способа определения критерия сортировки множества.

* *В параметре шаблона*, например:

set<int, greater<int> > coll;

В этом случае критерий сортировки является частью типа. Если критерий сортировки явно не задан, по умолчанию понимается less<>.



В выражении <int,greater<int> > между двумя угловыми скобками обяза- тельно должен присутствовать пробел, иначе компилятор не сможет правильно интерпретировать это выражение.

* *В параметре конструктора.* Этот способ применяется в случаях, когда необходимо определить несколько критериев сортировки для одного типа, а также для формирования критерия на стадии выполнения программы.

***Таблица 4.18.*** *Немодифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| c.size() | Возвращает количество элементов множества |
| c.empty() | Возвращает true, если множество пустое |

***Таблица 4.18*** *(окончание)*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| c.max\_size() | Возвращает максимальный размер множества |
| c1 == c2 | Проверяет равенство двух множеств |
| c1 != c2 | Проверяет неравенство двух множеств |
| c1 < c2 | Проверяет, что c1 < c2 |
| c1 > c2 | Проверяет, что c1 > c2 |
| c1 <= c2 | Проверяет, что c1 <= c2 |
| c1 >= c2 | Проверяет, что c1 >= c2 |
| c.count(elem) | Возвращает количество элементов со значением elem |
| c.find(elem) | Возвращает позицию первого элемента со значением  elem или end, если элемент не найден |
| c.lower\_bound(elem) | Возвращает позицию первого элемента со значением >= elem |
| c.upper\_bound(elem) | Возвращает последнюю позицию со значением > elem |
| c.equal\_range(elem) | Возвращает интервал (pair), в котором элементы равны  elem |
| c.begin() | Возвращает итератор первого элемента |
| c.end() | Возвращает итератор за последним элементом |
| c.rbegin() | Возвращает обратный итератор начала контейнера для перебора в обратном направлении |
| c.rend() | Возвращает обратный итератор конца контейнера для перебора в обратном направлении |

***Таблица 4.19.*** *Модифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| c1 = c2 | Присваивает c1 все элементы c2 |
| c1.swap(c2) | Меняет местами содержимое c1 и c2 |
| swap(c1,c2) | То же в форме функции |
| c.insert(elem) | Вставляет elem, возвращает позицию нового элемента |
| c.insert(pos,elem) | Вставляет elem, возвращает позицию нового элемента pos — определяет позицию, с которой следует начать по- иск позиции вставляемого элемента |

***Таблица 4.19*** *(окончание)*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| c.insert(beg,end) | Вставляет все элементы интервала [beg,end) |
| c.erase(elem) | Удаляет все элементы elem и возвращает количество уда- ленных элементов |
| c.erase(pos) | Удаляет элемент в позиции итератора pos |
| c.erase(beg,end) | Удаляет все элементы из интервала [beg,end) |
| c.clear() | Удаляет все элементы |

Для выполнения операций присваивания контейнеры должны быть одного типа, однако критерии сортировки могут быть разными. Листинг 4.26 — текст программы, в которой приведен пример использования множества.

**Листинг 4.26**

#include <iostream> #include <set> #include <iterator> using namespace std; void main()

{

int arr[] = {1, 5, 4, 2, 8, 3, 7, 6, 9 };

set<int> c, c1(arr, arr+9);

cout << "Size c1: " << c1.size() << endl;

cout << "Max Size c1: " << c1.max\_size() << endl; cout << "empty c?: " << c.empty() << endl;

c = c1;

cout << "set c: "; copy(c.begin(),c.end(),ostream\_iterator<int>(cout," ")); cout << endl;

if (c == c1) cout << "c == c1\n"; else cout << "c != c1\n"; set<int>::iterator it = c.find(4);

cout << "position 4 is: " << distance(c.begin(),it) << endl; c.insert(10);

c.erase(1);

cout << "set c: "; copy(c.begin(),c.end(),ostream\_iterator<int>(cout," ")); cout << endl;

}

Вывод:

Size c1: 9

Max Size c1: 1073741823 empty c?: 1

set c: 1 2 3 4 5 6 7 8 9

c == c1 position 4 is: 3

set c: 2 3 4 5 6 7 8 9 10

Для создания множества мы использовали массив целых чисел, причем мас- сив неупорядоченный. Тем не менее конструктор вполне корректно обрабо- тал этот массив и упорядочил данные. Проблема возникла, когда мы попыта- лись вычислить позицию найденного значения.



Для множества термин "позиция" может рассматриваться лишь в контексте за- данного упорядочения.

Поскольку элементы множества не занимают смежные области памяти, то для вычисления расстояния между данными невозможно использовать раз- ность итераторов (для множества эта операция не имеет смысла).



Расстояние здесь понимается как количество шагов, которые нужно сделать, чтобы из начальной точки прийти в конечную, учитывая критерий сортировки.

В этой ситуации можно воспользоваться функцией обработки расстояния между элементами контейнера distance(pos1,pos2). При использовании этой функции нужно учитывать, что pos2 должна быть достижима из pos1. Необ- ходимо также подключить заголовочный файл iterator.

**Контейнеры *Map, Multimap***

Элементами *отображения* и *мультиотображения* являются пары "ключ/зна- чение". Упорядочение элементов производится автоматически на основании критерия сортировки, применяемого к ключу. По умолчанию критерий сор- тировки <. Эти контейнеры отличаются только тем, что мультиотображения могут содержать дубликаты, а отображения — нет. Чтобы использовать ото- бражения и мультиотображения, необходимо включить в программу заголо- вочный файл:

#include <map> или #include <multimap>

Как и остальные контейнеры, отображения и мультиотображения определя- ются как шаблонные классы в пространстве имен std. Причем мультиотоб- ражения могут содержать одинаковые значения, а отображения — нет. В дальнейшем под отображением map будем понимать и мультиотображения multimap.

namespace std {

template <class Key, class T, class Compare = less<Key>, class Allocator = allocator<pair<const Key, T> > >

class map;

}

Как и все ассоциативные контейнеры, они реализованы в виде сбалансиро- ванного бинарного дерева. Множество и мультимножество, по сути, являют- ся частными случаями отображения и мультиотображения, соответственно, у которых ключ совпадает со значением. Таким образом, для отображения нельзя напрямую менять значение ключа, но прямая модификация его значе- ния разрешена. Однако, чтобы изменить значение ключа, необходимо этот элемент сначала удалить из отображения, затем вставить его с новым клю- чом.

Операции с отображениями и мультиотображениями рассмотрим по группам, составив соответствующие таблицы: конструкторы, немодифицирующие операции, модифицирующие операции, прямой доступ к элементам отобра- жения (табл. 4.20—4.23).

***Таблица 4.20.*** *Конструкторы*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| map<Key,T> c | Создает пустое отображение |
| map<Key,T> c(op) | Создает пустое отображение с критерием сортиров- ки op |
| map<Key,T> c1(c) | Создает копию отображения |
| map<Key,T> c(beg,end) | Создает отображение из элементов интервала  [beg,end) |
| map<Key,T> c(beg,end,op) | Создает отображение с критерием сортировки op из элементов интервала [beg,end) |

Как и для множества, существует два способа определения критерия сорти- ровки отображения.

* *В параметре шаблона*, например:

map<int, string, greater<int> > coll;

В этом случае критерий сортировки является частью типа. Если критерий сортировки явно не задан, по умолчанию понимается less<>.

* *В параметре конструктора*. Этот способ применяется в случаях, когда необходимо определить несколько критериев сортировки для одного типа, а также для формирования критерия на стадии выполнения программы.

***Таблица 4.21.*** *Немодифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| c.size() | Возвращает количество элементов отображения |
| c.empty() | Возвращает true, если отображение пустое |
| c.max\_size() | Возвращает максимальный размер отображения |
| c1 == c2 | Проверяет равенство |
| c1 != c2 | Проверяет неравенство |
| c1 < c2 | Проверяет, что c1 < c2 |
| c1 > c2 | Проверяет, что c1 > c2 |
| c1 <= c2 | Проверяет, что c1 <= c2 |
| c1 >= c2 | Проверяет, что c1 >= c2 |
| c.count(key) | Возвращает количество элементов с ключом key |
| c.find(key) | Возвращает позицию первого элемента с ключом key или  end |
| c.lower\_bound(key) | Возвращает позицию первого элемента со значением  >= key |
| c.upper\_bound(key) | Возвращает последнюю позицию с ключом > key |
| c.equal\_range(key) | Возвращает интервал (pair), в котором ключи == key |
| c.begin() | Возвращает итератор первого элемента |
| c.end() | Возвращает итератор за последним элементом |
| c.rbegin() | Возвращает обратный итератор первого элемента для перебора в обратном направлении |
| c.rend() | Возвращает обратный итератор последнего элемента для перебора в обратном направлении |

***Таблица 4.22.*** *Модифицирующие операции*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| c1 = c2 | Присваивает c1 все элементы c2 |
| c1.swap(c2) | Меняет местами содержимое c1 и c2 |
| swap(c1,c2) | То же в форме функции |
| c.insert(elem) | Вставляет elem, возвращает позицию нового элемента |
| c.insert(pos,elem) | Вставляет elem, возвращает позицию нового элемента. pos – определяет позицию, с которой следует начать поиск позиции вставляемого элемента |
| c.insert(beg,end) | Вставляет все элементы интервала [beg,end) |
| c.erase(elem) | Удаляет все элементы elem и возвращает количество уда- ленных элементов |
| c.erase(pos) | Удаляет элемент в позиции итератора pos |
| c.erase(beg,end) | Удаляет все элементы из интервала [beg,end) |
| c.clear() | Удаляет все элементы |



При использовании методов вставки и удаления необходимо учитывать, что элемент представляет пару "ключ/значение". Можно использовать шаблон pair<>(), например:

spr.insert(pair<string,string>(string("aaaa"),string("11"))); или использовать функцию make\_pair(), например: spr.insert(make\_pair( string("tttt"), string("22")));

***Таблица 4.23.*** *Прямой доступ к элементам отображения*

|  |  |
| --- | --- |
| **Операция** | **Назначение** |
| c[key] | Возвращает элемент с ключом key. Вставляет элемент, если его не существует |

В данном контексте отображения рассматриваются как *ассоциативные мас- сивы*, которые отличаются от обычных тем, что в качестве индекса использу- ется ключевое поле отображения. Индекс ассоциативного массива в принци- пе не может быть неверным, поскольку в случае отсутствия элемента он соз- дается автоматически при помощи конструктора по умолчанию. Что, кстати, может привести к случайному созданию записи.

Для примера использования отображения рассмотрим создание телефонного справочника. Здесь в качестве типа данных используем стандартный класс string. Организуем ввод данных с консоли и, для примера, вставим две запи- си. Текст программы приведен в листинге 4.27.

**Листинг 4.27. Создание телефонного справочника**

#include <string> #include <iostream> #include <map>

#include <utility> // Здесь описан шаблон pair< > using namespace std;

void main()

{

map<string, string> spr; string name, number;

cout << "Enter a name and number\n"; while(true)

{

getline(cin, name);

if (name.empty()) break; getline(cin, number);

spr[name] = number; // Добавление элемента контейнера

}

spr.insert(pair<string, string>(string("aaaa"),string("11"))); spr.insert(make\_pair( string("tttt"),string("22"))); map<string, string>::iterator it;

for (it = spr.begin(); it != spr.end(); it++)

cout << it->first << ' ' << it->second << endl;

}// it->first – ключ, it->second – значение

Вывод:

Enter a name and number aaaa 11

tttt 22

Обратите внимание на использование функции make\_pair(). Эта функция создает пару значений без явного указания типа, что упрощает текст про- граммы и не приводит к усложнению кода, поскольку компилятор автомати- чески оптимизирует такую конструкцию.

## Итераторы

*Итераторы* — это интеллектуальные указатели, служащие для перебора элементов. Поведение итератора зависит от контейнера, для которого он оп- ределен, и направлено на оптимизацию операций с конкретным контейнером. Они описываются в виде встроенного класса для каждого контейнера, поэто- му нет необходимости использовать специальный файл включений, и лишь некоторые, общие для всех итераторов функции, описаны в файле включений iterator. Для всех итераторов определены конструктор по умолчанию и ко- пирующий конструктор. По своим возможностям итераторы разделены на категории.

### Итераторы ввода

*Итератор ввода* позволяет перемещаться только в прямом направлении и поддерживает только чтение. Операции итераторов ввода приведены в табл. 4.24.

***Таблица 4.24.*** *Операции итераторов ввода*

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| \*iter | Обращение к элементу |
| iter->field | Обращение к полю или методу прочитанного элемента |
| ++iter | Передвижение вперед на 1 элемент (возвращает новую по- зицию) |
| iter++ | Передвижение вперед на 1 элемент (возвращает старую позицию) |
| iter1 == iter2 | Сравнение двух итераторов на равенство |
| iter1 != iter2 | Сравнение двух итераторов на неравенство |

### Итераторы вывода

*Итератор вывода* позволяет перемещаться только в прямом направлении, но поддерживает только запись. Таким образом, после операции записи элемент уже не доступен. Операции итераторов вывода приведены в табл. 4.25.

В листинге 4.28 представлен текст программы посимвольного чтения с кон- соли с входными итераторами.

***Таблица 4.25.*** *Операции итераторов вывода*

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| \*iter = value | Записывает value в позицию итератора |
| ++iter | Передвижение вперед на 1 элемент (возвращает новую пози- цию) |
| iter++ | Передвижение вперед на 1 элемент (возвращает старую пози- цию) |

|  |
| --- |
| **Листинг 4.28. Посимвольное чтение с консоли с входными итераторами** |

#include <iostream> #include <iterator> #include <vector> using namespace std; void main()

{

cin.unsetf(ios::skipws);// Отмена пропуска разделителей при вводе vector<char> str(20);

vector<char>::iterator it = str.begin(); istream\_iterator<char> in(cin);

// Читаем до нажатия клавиши Enter

while ((\*it++ = \*in) != '\n') ++in;

for (it = str.begin(); it != str.end(); it++) cout << \*it; cout << endl;

}

По сути дела, вся программа свелась к конструкции: \*it++ = \*in, где мы ко- пируем входной \*in символ в вектор. Наращивание входного итератора ++in осуществляем до символа '\n', который и завершает поток ввода. Здесь обя- зательно нужно запретить пропуск разделителей при вводе, иначе мы не сможем завершить ввод, поскольку символ перевода строки будет интерпре- тироваться как разделитель и игнорироваться во входном потоке. Выводим содержимое вектора на консоль при помощи обычной операции разадресации для итератора \*it.

### Прямые итераторы

*Прямые итераторы* представляют собой комбинацию итераторов ввода и вывода и при перемещении только в прямом направлении и позволяют осу- ществлять операции чтения и записи. Причем, доступ к элементу может

осуществляться неоднократно. Операции прямых итераторов приведены в табл. 4.26.

***Таблица 4.26.*** *Операции прямых итераторов*

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| \*iter | Обращение к элементу |
| iter->field | Обращение к полю или методу прочитанного элемента |
| ++iter | Передвижение вперед на 1 элемент (возвращает новую пози- цию) |
| iter++ | Передвижение вперед на 1 элемент (возвращает старую по- зицию) |
| iter1 == iter2 | Сравнение двух итераторов на равенство |
| iter1 != iter2 | Сравнение двух итераторов на неравенство |
| iter1 = iter2 | Присваивание итератора |

### Двунаправленные итераторы

*Двунаправленные итераторы* — это модификация прямых итераторов, для которой определены дополнительные операции для передвижения в обрат- ном направлении, приведенные в табл. 4.27.

***Таблица 4.27.*** *Дополнительные операции двунаправленных итераторов*

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| --iter | Передвижение назад на 1 элемент (возвращает новую позицию) |
| iter-- | Передвижение назад на 1 элемент (возвращает старую позицию) |

Рассмотрим, как при помощи двунаправленного итератора можно вывести содержимое контейнера в обратном направлении без использования обрат- ных итераторов. Необходимо лишь добавить одну строку в предыдущий пример:

for (it = str.end(); it > str.begin();) cout << \*(--it);

Необходимо здесь еще раз отметить, что операция сравнения it > str.begin() корректно работает только с вектором, поскольку лишь вектор занимает непрерывную область памяти.

### Итераторы произвольного доступа

*Итераторами произвольного доступа* называются двунаправленные итера- торы, обеспечивающие перемещение на произвольное количество элементов в обоих направлениях. Например, контейнер vector обеспечивает поддержку итераторов прямого доступа. Дополнительные операции итераторов прямого доступа приведены в табл. 4.28.

***Таблица 4.28.*** *Дополнительные операции итераторов прямого доступа*

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| iter[n] | Обращение к элементу с индексом n |
| iter += n | Смещение на n элементов вперед (назад, если n<0) |
| iter -= n | Смещение на n элементов назад (вперед, если n<0) |
| iter + n | Возвращает итератор n–го элемента вперед от текущей пози- ции |
| n + iter |
| iter - n | Возвращает итератор n–го элемента назад от текущей пози- ции |
| iter1 – iter2 | Возвращает расстояние между итераторами. Разумеется, итераторы должны принадлежать одному контейнеру, иначе операция не имеет смысла |
| iter1 < iter2 | Сравнение итераторов |
| iter1 > iter2 |
| iter1 <= iter2 |
| iter1 >= iter2 |

Вообще-то итератор контейнера vector является итератором произвольного доступа, и для него доступны все рассмотренные выше операции. Рассмот- рим, например, как вывести на консоль лишь четные символы, добавив также еще одну строку в предыдущий пример:

for (it = str.begin(); it < str.end(); it +=2) cout << \*it;

Здесь мы просто увеличиваем итератор it на 2.

### Обратные итераторы

Для передвижения по контейнеру в обратном направлении введены *обрат- ные итераторы*, которые имеют точно такой же синтаксис, что и прямые, но, в связи с особенностью реализации концепции полуоткрытого интервала

[beg, end), значение, на которое они ссылаются, смещены на одну позицию влево, как показано на рис. 4.6.



**Рис. 4.6.** Прямые и обратные итераторы контейнера

Описываются обратные итераторы, например, так:

vector<char>::reverse\_iterator rit = str.rbegin();

Иногда бывает необходимо преобразовать обратный итератор в прямой. Для такого преобразования существует метод base(). Например: it = rit.base(). Тестовая программа преобразования итераторов приведена в листинге 4.29.

|  |
| --- |
| **Листинг 4.29. Тестовая программа преобразования итераторов** |

#include <iostream> #include <algorithm> #include <vector> using namespace std; void main()

{

int m[] = { 1, 2, 3, 4, 5, 6, 7 };

vector<int> v(m, m+7);

vector<int>::iterator pos = find(v.begin(), v.end(), 4); cout << "pos: " << \*pos << endl; vector<int>::reverse\_iterator rpos(pos);

cout << "rpos: " << \*rpos << endl; pos = rpos.base();

cout << "rrpos: " << \*pos << endl;

}

Вывод:

pos: 4

rpos: 3

rrpos: 4

Функция find() находит позицию числа 4. Далее мы преобразуем прямой итератор в обратный, используя конструктор обратного итератора, который может принимать в качестве параметра прямой итератор.

vector<int>::reverse\_iterator rpos(pos);

После чего преобразуем обратный итератор в прямой методом base(). Ре- зультаты вывода показывают, что обратный итератор смещается на 1 пози- цию.

### Итераторы вставки

*Итераторы вставки* реализованы в виде отдельных классов и преобразуются просто в обычную операцию вставки. Для них имеет значение лишь операция присваивания:

insert\_iterator = value;

Определены еще три операции:

* \*iterator

 ++ iterator

* iterator++

но они реализованы в виде фиктивной операции, возвращающей исходный

iterator.

Различают три разновидности итераторов вставки (табл. 4.29), которые реа- лизуются соответствующими методами контейнера (вставка в конец, начало или произвольную позицию).

***Таблица 4.29.*** *Разновидности итераторов вставки*

|  |  |  |
| --- | --- | --- |
| **Название** | **Класс** | **Вызываемый метод** |
| Конечный итератор вставки | back\_insert\_iterator | push\_back(value) |
| Начальный итератор вставки | front\_insert\_iterator | push\_front(value) |
| Общий итератор вставки | insert\_iterator | insert(pos,value) |

Поскольку итераторы вставки реализованы в виде шаблонных классов, опи- сание переменных производится с указанием конкретного контейнера. Но и контейнер также реализован шаблонным классом, поэтому описание, как правило, содержит вложенные конструкции "< < > >", например:

back\_insert\_iterator<list<int> > it(lst);

Нужно напомнить, что между двумя символами "> >" необходимо ставить пробел, иначе компилятор будет интерпретировать это выражение, как опе- рацию >>.

Кроме классов, реализованы также и функции, обеспечивающие более про- стой способ создания итераторов вставки:

* back\_inserter(Container);
* front\_inserter(Container);
* inserter(Container, pos);

Эти функции возвращают соответствующий итератор, поэтому их примене- ние имеет несколько своеобразный синтаксис, например:

back\_inserter(Container) = value;

Пример программы использования итераторов вставки приведен в листин- ге 4.30.

|  |
| --- |
| **Листинг 4.30. Использование итераторов вставки** |

#include <iostream> #include <algorithm> #include <list> using namespace std; void main()

{

list<int> lst;

// Вставка при помощи итератора back\_insert\_iterator<list<int> > it(lst); it = 1;

it = 2;

// Операция вставки при помощи функции back\_inserter(lst) = 3;

inserter(lst, ++lst.begin()) = 4; list<int>::iterator r;

for (r = lst.begin(); r != lst.end(); ++r) cout << \*r << endl;

}

Вывод:

1

4

2

3

### Вспомогательные функции итераторов

Набор функций, обеспечивающий дополнительный сервис при работе с ите- раторами, размещен в файле включений iterator.

**Функция перемещения итератора *advance()***

void advance(Iterator& pos, Dist n)

Обеспечивает смещение итератора на n позиций относительно текущего по- ложения. Шаблонный тип Dist обычно является целым типом. Функция обеспечивает оптимальную скорость выполнения операций, так для контей- нера vector она выполняет операцию pos+=n; а для контейнера list — n раз выполняет операцию ++pos (или --pos). Нужно только иметь в виду, что функция не контролирует выход за пределы контейнера.

**Расстояние между итераторами *distance()***

Dist distance(Iterator pos1, Iterator pos2)

Возвращает расстояние между итераторами ввода pos1 и pos2. Разумеется, оба итератора должны принадлежать одному контейнеру, и итератор pos2 должен быть достижим от позиции итератора pos1.

**Перестановка элементов функцией *iter\_swap()***

void iter\_swap(Iterator pos1, Iterator pos2)

Функция переставляет местами элементы, на которые ссылаются итераторы

pos1 и pos2. Описание помещено в файл включений algorithm.

Пример программы с использованием вспомогательных функций итераторов приведен в листинге 4.31.

|  |
| --- |
| **Листинг 4.31. Использование вспомогательных функций итераторов** |

#include <iostream> #include <algorithm> #include <iterator> #include <list> using namespace std; void main()

{

list<int> lst;

for (int i = 0; i < 10; i++) lst.push\_back(i);

// Обмен местами первого и последнего элемента списка iter\_swap(lst.begin(), --lst.end());

copy(lst.begin(), lst.end(), ostream\_iterator<int>(cout," "));

// Размер контейнера – расстояние между его началом и концом

cout <<endl<<"Distance : "<<distance(lst.begin(),lst.end())<<endl; list<int>::iterator it = lst.begin();

// Сдвиг итератора на 5 позиций вперед advance(it, 5);

cout << "Element num 5 is: " << \*it << endl;

}

Вывод:

9 1 2 3 4 5 6 7 8 0

Distance : 10

Element num 5 is: 5

## Специальные контейнеры

Стандартная библиотека включает в себя так называемые *контейнерные адаптеры*, т. е. специальные классы, адаптирующие стандартные контейнеры для выполнения специфических задач. К таким адаптерам относятся: стеки, очереди и приоритетные очереди.

**Контейнер *Stack***

*Стек* представляет собой шаблонный класс (stack), работающий по принци- пу "первым пришел — последним вышел", определенный в файле stack в стандартной области имен std:

template <class T, class Container = deque<T> > class stack;

По умолчанию класс определен как адаптер контейнера дек. Это сделано из тех соображений, что дек более эффективно использует память и обходится без копирования всех элементов контейнера при перераспределении памяти, поскольку занимает несколько несмежных блоков памяти и в случае необхо- димости копирует лишь один из своих блоков. Однако стек можно объявить с любым другим последовательным контейнером, поддерживающим функ- ции: back(), push\_back(), pop\_back(), например, на базе контейнера vector или list.

stack<int> st;

stack<int, vector<int> > sv;

В *главе 2* мы строили собственный класс stack. Стандартный класс, по своим функциональным возможностям, мало чем от него отличается.

Пример текста программы, иллюстрирующей работу со стеком, приведен в листинге 4.32.

|  |
| --- |
| **Листинг 4.32. Работа со стеком** |

#include <iostream> #include <stack> using namespace std; void main()

{

stack<int> st;

for (int i = 0; i < 10; i++) st.push(i); // Заполнение стека while (!st.empty()) // Цикл "пока стек не пустой"

{

cout << st.top() <<endl; // Вывод элемента на консоль st.pop(); // Выталкивание "верхнего элемента"

}

}

**Методы класса *stack***

Класс stack строится не для удобства работы программиста, а для макси- мальной эффективности этого контейнера, поэтому он имеет минимальное количество методов, из них основные три:

* push() — вставляет элемент в стек;
* pop() — удаляет элемент из стека;
* top() — возвращает верхний элемент стека.

**Контейнер *Deque***

*Очередь* представляет собой шаблонный класс, определенный как адаптер контейнера дек и работающий по принципу "первым пришел — первым вы- шел" (рис. 4.7):

template<class T, class Container = deque<T> > class queue;

**Рис. 4.7.** Схема функционирования очереди



Точно так же, как и для стека, для организации очереди можно использовать любой последовательный контейнер, например список:

queue <int, list<int> > line;

Для использования очереди необходимо включить заголовочный файл queue. Рассмотрим в качестве примера предыдущую задачу, где мы вместо стека используем очередь (листинг 4.33).

**Листинг 4.33. Работа с очередью**

#include <iostream> #include <queue> using namespace std; void main()

{

queue<int> line;

for (int i = 0; i < 10; i++) line.push(i); while (!line.empty())

{

cout << line.front() <<endl; line.pop();

}

}

Так же, как и в стеке, здесь имеются методы:

* push() — вставляет элемент в очередь;
* pop() — удаляет элемент в конце очереди;
* back() — возвращает последний элемент очереди (тот, который был вве- ден позднее других);
* front() — возвращает первый элемент очереди (тот, который был введен раньше всех).

Принципиальное отличие этих программ (листинг 4.32—4.33) заключается лишь в том, что последняя программа выводит содержимое контейнера в по- рядке заполнения, а первая выводит данные в обратном порядке.

**Контейнер *Priority\_queue***

Шаблонный класс *приоритетная очередь* определен как адаптер контейнера vector и реализует очередь, в которой последовательность чтения элементов определяется их приоритетами. В отличие от очереди, следующим элементом чтения является не первый вставленный элемент, а элемент с максимальным

приоритетом. Приоритет здесь определяется критерием сортировки, который является третьим необязательным параметром в объявлении приоритетной очереди (по умолчанию элементы сравниваются с оператором <). Определе- ние класса также размещено в файле включений queue.

template< class T,

class Container = vector<T>,

class Compare = less <typename Container::value\_type> > class priority\_queue;

Методы приоритетной очереди:

* push() — вставляет элемент в приоритетную очередь;
* top() — возвращает следующий элемент приоритетной очереди;
* pop() — удаляет элемент из приоритетной очереди.

Так, если взять предыдущую задачу, в которой заменить очередь на приори- тетную очередь, мы увидим, что данные упорядочены в порядке убывания (листинг 4.34).

**Листинг 4.34. Работа с приоритетной очередью**

#include <iostream> #include <queue> using namespace std; void main()

{

priority\_queue<int> line;

for (int i = 0; i < 10; i++) line.push(i); while (!line.empty())

{

cout << line.top() <<endl; line.pop();

}

}

Здесь нам пришлось заменить и метод для вывода данных top().

**Класс *string***

В стандартную библиотеку STL входит и класс string, который, конечно же, имеет больше возможностей, чем разработанный нами вариант класса работы со строками. Определение его размещено в файле включений string. При создании этого класса использована та же идея, что и при создании нашего

собственного класса. Символы хранятся в динамическом массиве типа char без завершающего символа строки, а размер массива хранится в отдельной переменной. Память под строку выделяется по мере необходимости.

Рассмотрим основные методы класса string: конструкторы, функции преоб- разования строк и С-строк, функции размера и емкости, операторы сравне- ния, присваивание, склеивание строк, вставка и удаление символов, замена символов, поиск, выделение подстроки, получение итераторов (табл. 4.30— 4.40).



На самом деле, класс string — это реализация шаблонного класса

basic\_string<> для типа char.

***Таблица 4.30.*** *Конструкторы*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| string s | Создает пустую строку |
| string s(str) | Создает копию строки str |
| string s(str,index) | Создает копию подстроки str, начиная с индекса  index |
| string s(str,index,len) | Создает копию подстроки str, начиная с индекса  index, размером не более len символов |
| string s(C\_str) | Создает строку, инициализированную С-строкой, т. е. символьным массивом с завершающим симво- лом '\0' |
| string s(char\_m,len) | Создает строку, инициализированную len символа- ми массива char\_m |
| string s(n,ch) | Создает строку из n символов ch |
| string s(beg,end) | Создает строку, инициализированную символами интервала [beg,end) |

***Таблица 4.31.*** *Функции преобразования строк и С-строк*

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| data() | Возвращает содержимое строки в виде массива симво- лов |
| c\_str() | Возвращает содержимое строки в виде С-строки |
| copy(buff, N,[index]) | Создает копию N символов строки в buff, начиная с индекса index |

***Таблица 4.32.*** *Функции размера и емкости*

|  |  |
| --- | --- |
| **Функции** | **Описание** |
| size() | Возвращает количество символов в строке |
| length() |
| empty() | Проверка пустой строки (s.size() == 0) |
| capacity() | Возвращает размер выделенной под строку памяти |
| reserve([N]) | Увеличивает емкость строки, по крайней мере, до N симво- лов. Без параметров приводит емкость к текущему размеру |
| resize(N[,char\_c]) | Изменяет размер строки до N, присоединяя или удаляя символы в конце строки. При увеличении размера строка заполняется символом char\_c, а при отсутствии второго параметра — символом '\0' |

Текст программы, в которой показан пример использования переменных класса string, приведен в листинге 4.35.

|  |
| --- |
| **Листинг 4.35. Использование переменных класса string** |

#include <string> #include <iostream> using namespace std; void main()

{

string s1, s2("text"), s3(10,'\*'); const char \*p = s3.data();

for (int i = 0; i < s3.size(); i++) cout << p[i]; cout << endl;

cout << "capacity s1: "<< s1.capacity() << endl; cout << "capacity s2: "<< s2.capacity() << endl; s2.reserve(100);

cout << "capacity new: "<< s2.capacity() << endl; s2.resize(200);

cout << "capacity resize: "<< s2.capacity() << endl;

}

Вывод:

\*\*\*\*\*\*\*\*\*\* capacity s1: 0

capacity s2: 31

capacity new: 127

capacity resize: 223

Что интересно, память выделяется блоками, кратными 32 байтам (без одного байта). Следует также обратить внимание на то, что, в отличие от классиче- ского С, в С++ для строковых литералов используется тип const char\*, по- этому мы должны были именно так описать указатель для возвращаемого значения метода data():

const char \*p = s3.data();

В следующей таблице (табл. 4.33) рассмотрим операторы сравнения.

***Таблица 4.33.*** *Операторы сравнения*

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| == |  |
| !=  <  >  <= | Сравнение производится по лексикографическому критерию. В ка- честве операндов в левой или правой части выражения могут участ- вовать и С-строки |
| >= |  |

Кроме операторов сравнения, в классе string реализован и метод compare(), который осуществляет сравнение текущей строки с образцом, в качестве ко- торого может выступать как строка string, так и С-строка. Возвращаемое значение: 0 — если строки равны; отрицательное число, если \*this < str; и положительное число, если \*this>str.

int compare([idx, len,] string\_str[, str\_ind, str\_len])

сравнивается не более символов len текущей строки, начиная с индекса idx с str\_len символами строки string\_str, начиная с символа str\_ind. Обяза- тельным здесь является лишь один параметр: string\_str.

И аналогичная конструкция для С-строки:

int compare([idx, len,] С\_str[,str\_len])

С-строка всегда сравнивается с начала.

Для обращения к символам строки перегружен оператор [], который возвра- щает текущий символ, а также реализован метод at(idx), который осуществ- ляет проверку выхода за пределы строки.

Например:

char ch = s2[2];

. . .

ch = s2.at(3);

***Таблица 4.34.*** *Присваивание*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| = | Оператор присваивания перегружен как для строки string, так и для С-строки. Возможно многократное присваивание |
| assign(string\_str[,idx,len]) | Присваивает не более len символов строки  string\_str, начиная с индекса idx |
| assign(C\_str[,len]) | Присваивает len символов из символьного мас- сива C\_str |
| assign(N,char\_c) | Присваивает N символов char\_c |
| swap(string\_str) | Обменивает содержимое \*this и string\_str |

Метод swap() реализован также в виде функции: swap(str1, str2).

Для демонстрации работы рассмотренных методов добавим к рассмотренной ранее программе несколько строк:

s1 = s2; swap(s2,s3);

if ( s1 == s3) cout << "Strings are equal\n";

Получим:

Strings are equal

***Таблица 4.35.*** *Склеивание строк*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| += | Оператор присоединения перегружен для строки string, С-строки и отдельного символа. Возвращает \*this |
| + | Оператор "склеивания" + перегружен для стро- ки string, С-строки и отдельного символа, как для первого, так и для второго операнда. Воз- вращает полученную строку |
| append(string\_str[,idx,len]) | Присоединяет не более len символов строки string\_str, начиная с индекса idx. Возвра- щает \*this |
| append(C\_str[,len]) | Присоединяет len символов из символьного массива C\_str. Возвращает \*this |
| append(beg, end) | Присоединяет все символы интервала [beg, end). Возвращает \*this |

Добавим еще несколько строк в наш пример:

s1.append(s2,0,3); s1 += "12345";

s1 += '!';

cout << s1 << endl;

Получим:

text\*\*\*12345!

***Таблица 4.36.*** *Вставка и удаление символов*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| insert(idx, string\_str [,str\_idx, str\_len]) | Вставляет фрагмент строки string\_str дли- ной str\_len с позиции str\_idx в текущую строку, начиная с позиции idx. Если два по- следних параметра отсутствуют, строка встав- ляется целиком |
| insert(idx,С\_str[,len]) | Вставляет len символов символьного массива или С-строки с позиции idx |
| insert(idx, N, char\_c) | Вставляет N символов char\_c с позиции idx |
| insert(pos,[N,] char\_c) | Вставляет N символов char\_c перед симво- лом, на который указывает итератор pos |
| insert(pos, beg, end) | Вставляет все символы интервала [beg, end) перед символом, на который указывает итератор pos |
| clear()  erase() | Очистка строки |
| erase(idx[,len]) | Удаляет len символов с позиции idx, или до конца строки, если второй параметр отсутст- вует |
| erase(pos) | Удаляет символ в позиции итератора pos |
| erase(beg, end) | Удаляет все символы интервала [beg,end) |

При вставке символов строка "раздвигается", а при удалении "сжимается", заполняя освободившиеся позиции символами справа.

Добавим небольшой фрагмент к нашей программе и посмотрим, что полу- чится.

s1 = "0123";

s2 = "abcd";

s1.insert(2,s2); cout << s1 << endl;

s2.insert(1,2,'#'); // Добавим 2 символа # cout << s2 << endl;

s1.erase(2,4); // Удалим подстроку "abcd" из s1 cout << s1 << endl;

Вывод:

01abcd23 a##bcd 0123

***Таблица 4.37.*** *Замена символов*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| replace(idx, len, string\_str [,str\_idx, str\_len]) | Заменяет не более len символов теку- щей строки с позиции idx на фрагмент строки string\_str с позиции str\_idx не более str\_len символов |
| replace(idx, len, С\_str [,str\_len]) | Заменяет не более len символов теку- щей строки на str\_len символов С-стро- ки или символьного массива |
| replace(idx, len, N, char\_c) | Заменяет не более len символов теку- щей строки с позиции idx N экземпляра- ми символа char\_c |
| peplace(beg,end,string\_str) | Заменяет все символы интервала [beg, end) символами строки string\_str |
| replace(beg, end, С\_str [,str\_len]) | Заменяет все символы интервала [beg, end) на str\_len символов С-строки или символьного массива |
| replace (beg,end,N,char\_c) | Заменяет все символы интервала [beg, end) на N экземпляров символа char\_c |
| replace (beg, end, newbeg, newend) | Заменяет все символы интервала [beg, end) символами интервала [newbeg, newend) |

Добавим к нашему примеру еще пару строк:

s1.replace(1,2,1,'&'); // Заменим 2 символа s1, начиная с индекса 1 cout << s1 << endl; // одним символом '&'

Вывод: 0&3

Как видно из рассмотренного примера, если символов для замены предлага- ется меньше, чем имеется в исходной строке, то строка просто "сжимается".

***Таблица 4.38.*** *Поиск*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| find(char\_c[,idx]) | Поиск первого вхождения в строке симво- ла char\_c, начиная с индекса idx |
| rfind(char\_c[,idx]) | Поиск последнего вхождения в строке символа char\_c, начиная с индекса idx |
| find(string\_str[,idx]) | Поиск первого вхождения подстроки  string\_str, начиная с индекса idx |
| rfind(string\_str[,idx]) | Поиск последнего вхождения подстроки  string\_str, начиная с индекса idx |
| find(C\_str[[,idx][,len]]) | Поиск первого вхождения С-строки C\_str, начиная с индекса idx или символьного массива размером len |
| rfind(C\_str[[,idx][,len]]) | Поиск последнего вхождения С-строки C\_str, начиная с индекса idx или сим- вольного массива размером len |
| find\_first\_of(string\_str [,idx]) | Поиск первого символа, который входит в строку string\_str, начиная с индекса idx |
| find\_first\_not\_of(string\_str [,idx]) | Поиск первого символа, который не вхо- дит в строку string\_str, начиная с ин- декса idx |
| find\_first\_of(C\_str [[,idx] [,len]]) | Поиск первого символа, который входит в C-строку C\_str, начиная с индекса idx |
| find\_first\_not\_of(C\_str [,idx] [,len]]) | Поиск первого символа, который не вхо- дит в C-строку C\_str, начиная с индекса idx |
| find\_first\_of (char\_c[,idx]) | Поиск первого символа, равного char\_c, начиная с индекса idx |
| find\_first\_not\_of (char\_c [,idx]) | Поиск первого символа, не равного  char\_c, начиная с индекса idx |
| find\_last\_of(string\_str [,idx]) | Поиск последнего символа, который вхо- дит в строку string\_str, начиная с ин- декса idx |

***Таблица 4.38*** *(окончание)*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| find\_last\_not\_of(string\_str [,idx]) | Поиск последнего символа, который не входит в строку string\_str, начиная с индекса idx |
| find\_last\_of(C\_str [[,idx] [,len]]) | Поиск последнего символа, который вхо- дит в C-строку C\_str, начиная с индекса idx |
| find\_last\_not\_of(C\_str [,idx] [,len]]) | Поиск последнего символа, который не входит в C-строку C\_str, начиная с ин- декса idx |
| find\_last\_of (char\_c[,idx]) | Поиск последнего символа, равного  char\_c, начиная с индекса idx |
| find\_last\_not\_of (char\_c,idx]) | Поиск последнего символа, не равного  char\_c, начиная с индекса idx |



Для всех функций поиска, при отсутствии второго параметра, поиск осуществ- ляется с конца строки.

Если метод имеет три параметра, то третий параметр интерпретируется как размер символьного массива, в этом случае символ '\0' не имеет особой ин- терпретации.

Все функции возвращают позицию найденного символа или начала подстро- ки типа size\_type, который является просто беззнаковым целым типом. Если поиск закончился неудачей, то возвращаемое значение string::npos(). (Это беззнаковое целое равное -1. Рекомендуется сравнивать с константой -1.)

***Таблица 4.39.*** *Выделение подстроки*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| substr([idx][,len]) | Возвращает подстроку из не более len символов, начиная с индекса idx. Без параметров возвращает копию строки |

***Таблица 4.40.*** *Получение итераторов*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| begin() | Возвращает итератор начала строки |
| end() | Возвращает итератор конца строки |

***Таблица 4.40*** *(окончание)*

|  |  |
| --- | --- |
| **Выражение** | **Описание** |
| rbegin() | Возвращает обратный итератор начала строки |
| rend() | Возвращает обратный итератор конца строки |

Переменные типа string поддерживают итераторы произвольного доступа, поскольку занимают непрерывную область памяти. Необходимо только учи- тывать, что после операций вставки/замены возможно неявное перераспреде- ление памяти, в связи с чем, после этих операций значение итераторов необ- ходимо переопределить.

### Функции ввода/вывода

В классах потокового ввода/вывода на консоль istream и ostream, а также и в файл ifstream и ofstream перегружены операторы >> и << для экземпляров класса string и мы можем осуществлять их ввод и вывод, как простых пере- менных, например:

string s1; cin >> s1;

cout << s1 << endl;

Рассмотрим, в качестве примера, следующую задачу.

Дан текстовый файл в формате MS DOS, необходимо вывести на консоль все встречающиеся слова, причем, если слова повторяются, выводить слово только один раз.

Для решения этой задачи лучше всего использовать множество, поскольку по своей структуре этот контейнер будет игнорировать повторяющиеся значе- ния. Но здесь возникает вопрос о критерии сортировки. Если мы выберем критерий сортировки по умолчанию less<>, то в списке будут присутствовать все вариации одного слова с разными окончаниями, поэтому мы создадим функциональный объект, который и будем использовать в качестве более слабого критерия. Будем считать, что вообще-то спорно, слова одинаковыми, если они совпадают на 80% своей длины.

Прежде чем мы будем помещать слово в контейнер, отбросим завершающие знаки пунктуации и исключим однобуквенные слова. Текст программы при- веден в листинге 4.36.

**Листинг 4.36. Задача "Глоссарий"**

#include <string> #include <fstream> #include <iostream> #include <set> #include <algorithm> using namespace std;

const double criterion = 0.8;

*class compare*

{

public: bool operator () (const string& s1, const string& s2) const

{

int l1 = s1.size(); int l2 = s2.size();

int min = (l1 < l2)? l1: l2; int max = (l1 > l2)? l1: l2; int k = max\*criterion+0.5; if (k > min) k = min;

for (int i = 0; i < k; i++)

if (s1[i] < s2[i]) return true;

else if (s1[i] > s2[i]) return false; return false;

}

};

void main()

{

string::size\_type k; string str;

set<string,compare> glossary; ifstream in("text.txt");

if (in)

{

while (in >> str)

{

k = str.find\_last\_of(".,;!?");

if ( k != -1) str.erase(k); if (str.size() == 1) continue; glossary.insert(str);

}

copy(glossary.begin(), glossary.end(), ostream\_iterator<string>(cout,"\n"));

}

else cout << "File not found\n";

}

Анализируя полученный список слов, можно увидеть, что сортировка в кон- тейнере осуществляется в лексикографическом порядке, т. е. все слова, начи- нающиеся с прописной буквы, оказались в начале списка, затем следуют сло- ва, начинающиеся со строчных букв. Если нас не устраивает такой порядок сортировки, то следует написать более совершенный функциональный объ- ект для определения необходимого критерия сортировки.

## Заключение

Стандартная библиотека STL содержит и другие компоненты, например классы: "умных указателей" auto\_ptr, битовых полей bitset, комплексных чисел complex, массивов значений valarray и т. д., а также множество сервис- ных функций. Но мы ограничимся пока рассмотренным выше материалом, полагая, что для вводного курса этого достаточно, а когда нам потребуются дополнительные возможности, мы вернемся к библиотеке STL.

## Вопросы к главе

1. Состав стандартной библиотеки STL.
2. Последовательные контейнеры. Конструкторы. Основные операции и особенности реализации.
3. Специальные контейнеры: стек, очередь, приоритетная очередь.
4. Ассоциативные контейнеры. Особенности реализации.
5. Итераторы, как интеллектуальные указатели. Доступ к элементам кон- тейнера через итераторы.
6. Классификация итераторов. Операции с контейнером через итераторы.
7. Итераторы вставки.
8. Вспомогательные функции итераторов: advance(), distance(), iter\_swap().
9. Алгоритмы поиска, возвращаемое значение.
10. Функциональные объекты. Три формы реализации. Адаптеры функцио- нальных объектов.
11. Алгоритмы копирования. Использование потоковых итераторов.
12. Алгоритмы преобразования.
13. Алгоритмы сортировки.
14. Алгоритмы упорядоченных интервалов.
15. Численные алгоритмы.
16. Особенности реализации класса string. Основные операции.

## Задание для самостоятельной работы

1. Определить в программе массив из 10 чисел типа double. Создать вектор из этого набора чисел и отсортировать его по возрастанию. Используя стандартные алгоритмы, построить вектор, координаты которого являют- ся квадратами исходных координат. Вычислить сумму координат обоих векторов, результаты вывести на консоль.
2. Решить ту же задачу, используя в качестве исходных данных массив чи- сел из текстового файла.
3. Прочесть текстовый файл и поместить как С-строки в контейнер vector. Отсортировать строки по алфавиту и вывести на консоль и в файл с тем же именем и расширением srt.
4. Решить ту же задачу, используя для хранения строки класс string.
5. Преобразовать предыдущую задачу, используя контейнеры deque, list.
6. В предыдущей задаче найти количество символов, слов и строк в тексто- вом файле. Вывести отчет на консоль.
7. Для набора целых чисел из текстового файла построить упорядоченный список всех встречающихся значений, найти частоты их появления. Ре- зультаты вывести на консоль.
8. Имеется текстовый файл с заданным набором чисел. Написать програм- му, которая заполняет контейнер (vector, deque, list) из текстового файла и, используя возможности стандартной библиотеки, вычисляет:

 количество чисел;  среднее значение;

 размах выборки (max–min);

Набор чисел, отсортированный по убыванию, вывести на консоль.

1. Организовать формирование двоичного файла из имеющегося текстового с набором числовых данных целого типа. Решить предыдущую задачу для двоичного файла.
2. Для предыдущей задачи разбить диапазон значений на 10 равных интер- валов и подсчитать количество значений, попавших в эти диапазоны. Найти диапазон с максимальным и минимальным заполнением. Результа- ты вывести на консоль.

**Г ЛАВА 5**

# Организация оконного интерфейса

Стиль программирования Windows-приложений принципиально отличается от того, который сложился в операционных системах раннего поколения. На- пример, в MS DOS именно программа является инициатором взаимодействия с операционной системой, а для организации обработки системных событий приходилось прилагать дополнительные усилия, вплоть до переустановки векторов прерываний. Совсем иначе дело обстоит в Windows, где, наоборот, именно операционная система обращается к программе. В связи с такой идеологией построения операционной системы, программа должна ждать по- сылки сообщения операционной системы и лишь после его получения вы- полнить определенные действия, затем вновь перейти в режим ожидания очередного сообщения. На рис. 5.1 схематично изображена диаграмма ти- пичной Windows-программы.



**Рис. 5.1.** Структура Windows-программы

Windows генерирует множество различных сообщений, которые направляют- ся программе — например, сообщения о нажатии кнопки мыши или клавиши на клавиатуре. Если программа не обрабатывает какие-то сообщения, опера- ционная система реагирует на них стандартным способом, так что задачей программиста является обработка лишь тех сообщений, которые необходимы для работы конкретной программы.

Следующим существенным отличием Windows-программы является возмож- ность ее непосредственного обращения к ресурсам Windows, используя функции *Программного Интерфейса Приложений* (Application Program Inter- face, API). Совсем иначе в Windows осуществляется и вывод на экран мони- тора. Дело в том, что эта операционная система изначально строилась как многозадачная, а в этом случае должна исключаться возможность монополь- ного захвата одной из программ устройства вывода, поэтому весь вывод на консоль должен осуществляться через оконные процедуры, а операционная система распределяет ресурс между окнами. Эта идеология реализована та- ким образом, что оконная функция вызывается не из программы, а операци- онной системой. Когда программа начинает выполняться, она должна опре- делить и зарегистрировать класс окна, с которым и будет взаимодействовать операционная система.



Здесь термин "класс окна" использован в смысле "тип окна".

Причем регистрация окна не означает его отображения. Для этого нужно вы- звать специальную функцию отображения окна.

Прежде чем переходить к детальному обсуждению структуры Windows- программы, рассмотрим типы данных, используемых в стандартных библио- теках. Так сложилось, что разработчики программного обеспечения под Windows ввели множество собственных типов данных, являющихся аналога- ми стандартных типов, но их использование стало стандартом де-факто при программировании Windows-приложений. Приведем наиболее типичные обо- значения (табл. 5.1).

Большая группа типов данных, название которых начинается с H, — это деск- рипторы (описатели), например HANDLE. На самом деле они являются целыми числами типа int и используются операционной системой в качестве индек- сов. Если же название типа начинается с LP — это указатель соответствующе- го типа, например, LPWORD — это WORD\*.

Еще одной особенностью Windows-программ является использование функ- ций с обратным порядком следования параметров вызова (как это принято в языке Pascal). В классическом С для вызова таких функций использовалось служебное слово pascal, здесь также введен его синоним CALLBACK. Для обо-

значения функций с прямым порядком передачи параметров в классическом С может быть использован модификатор cdecl, но его обычно опускают, по- скольку он определен по умолчанию.



В Windows все API-функции являются функциями с обратным порядком следо- вания параметров, но для них используется служебное слово WINAPI.

***Таблица 5.1.*** *Имена типов данных, используемых в Windows*

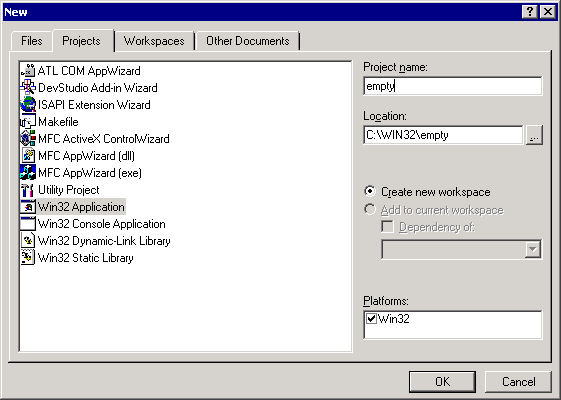
|  |  |
| --- | --- |
| **Тип** | **Аналог в С++** |
| DWORD COLORREF | unsigned long |
| FLOAT | float |
| INT | int |
| LONG | long |
| LPARAM LRESULT | long |
| UINT | unsigned int |
| WORD ATOM | unsigned short |
| BOOL | int |
| BYTE | unsigned char |

## Каркас Windows-приложения

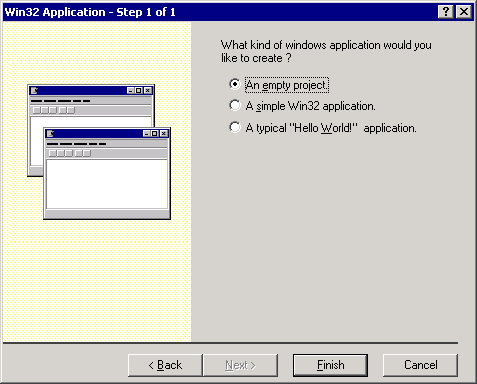
Структура Windows-программы должна быть стандартизирована для того, чтобы пользователь, загрузив любую программу, имел примерно одинаковый интерфейс. Для достижения этой цели в среде разработки Visual C++ созданы специальные мастера, но вначале рассмотрим создание проекта "вручную".

Для того чтобы иметь возможность работать с Windows-окнами, заготовка, или каркас Windows-приложения, должна выполнить некоторые стандартные действия.

1. Определить *класс окна*.
2. Зарегистрировать окно.
3. Создать окно данного класса.



**Рис. 5.2.** Создание 32-битного Windows-приложения



**Рис. 5.3.** Создание "пустого" проекта

1. Отобразить окно.
2. Запустить цикл обработки сообщений.

Рассмотрим технику создания простейшего 32-разрядного Windows-прило- жения. Последовательность действий по созданию "пустого" проекта в среде MS Visual C++ 6.0 показана на рис. 5.2, 5.3. В результате этих действий будет создана пустая папка с именем проекта. Выполнив команду меню **File | New**, на вкладке **Files** добавим **C++ Sourse File** с именем, например, main, после чего можно вводить приведенный далее текст (листинг 5.1), представляющий каркас Windows-приложения.

**Листинг 5.1. Простейшая Windows-программа, или каркас Windows-приложения**

#include <windows.h>

LRESULT CALLBACK WindowsFunc(HWND, UINT, WPARAM, LPARAM);

char WinName[] = "MainFrame";

int WINAPI WinMain(HINSTANCE This, // Дескриптор текущего приложения HINSTANCE Prev, // В современных системах всегда 0

LPSTR cmd, // Командная строка

int mode) // Режим отображения окна

{

HWND hwnd; // Дескриптор главного окна программы MSG msg; // Структура для хранения сообщения WNDCLASS wc; // Класс окна

// Определение класса окна wc.hInstance = This;

wc.lpszClassName = WinName; // Имя класса окна wc.lpfnWndProc = WindowsFunc; // Функция окна wc.style = 0; // Стиль по умолчанию

wc.hIcon = LoadIcon(NULL,IDI\_APPLICATION); // Стандартная иконка wc.hCursor = LoadCursor(NULL,IDC\_ARROW); // Стандартный курсор wc.lpszMenuName = NULL; // Нет меню

wc.cbClsExtra = 0; // Нет дополнительных данных класса wc.cbWndExtra = 0; // Нет дополнительных данных окна

// Заполнение окна белым цветом

wc.hbrBackground = (HBRUSH)GetStockObject(WHITE\_BRUSH); if(!RegisterClass(&wc)) return 0; // Регистрация класса окна

// Создание окна

hwnd = CreateWindow(WinName, // Имя класса окна "Каркас Windows-приложения", // Заголовок окна WS\_OVERLAPPEDWINDOW, // Стиль окна CW\_USEDEFAULT, // x

CW\_USEDEFAULT, // y Размеры окна

CW\_USEDEFAULT, // Width CW\_USEDEFAULT, // Height

HWND\_DESKTOP, // Дескриптор родительского окна NULL, // Нет меню

This, // Дескриптор приложения

NULL); // Дополнительной информации нет ShowWindow(hwnd, mode); // Показать окно

// Цикл обработки сообщений while(GetMessage(&msg, NULL, 0, 0))

{

TranslateMessage(&msg);// Функция трансляции кодов нажатой клавиши DispatchMessage(&msg); // Посылает сообщение функции WindowsFunc()

}

}

// Оконная функция вызывается операционной системой

// и получает сообщения из очереди для данного приложения LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message,

WPARAM wParam, LPARAM lParam)

{ // Обработчик сообщений switch(message)

{

case WM\_DESTROY : PostQuitMessage(0); break; // Завершение программы

// Обработка сообщения по умолчанию

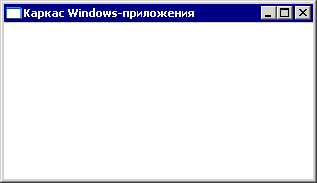
default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}

Программа не делает ничего полезного, поэтому, запустив ее на выполнение, мы получим изображенное на рис. 5.4 пустое окно, имеющее заголовок и на- бор стандартных кнопок.



**Рис. 5.4.** Результат выполнения нашей первой Windows-программы

### Исследование программы

Сейчас подробно рассмотрим текст нашей программы. Первая строка содер- жит единственный файл включений, который обязательно присутствует во всех Windows-программах.

#include <windows.h>

Основа файла windows.h состоит из вызова четырех файлов включений: windef.h, winbase.h, wingdi.h, winuser.h, а также нескольких дополнительных файлов, в которых размещены основные определения и описание API- функций.

Далее следует прототип оконной функции:

LRESULT CALLBACK WindowsFunc(HWND, UINT, WPARAM, LPARAM);

Отличительной чертой оконной функции является обратный порядок вызова параметров, что связано с некоторыми особенностями вызовов от операцион- ной системы. Эта функция регистрируется в системе головной программой, а ее вызов осуществляет именно операционная система, когда требуется обра- ботать системное сообщение.

Затем на глобальном уровне описывается текстовый массив:

char WinName[] = "MainFrame";

Массив содержит С-строку с именем приложения и используется операцион- ной системой для его идентификации. Имя может быть произвольным, в ча- стности, содержать кириллический текст. Не является обязательным описа- ние WinName на глобальном уровне, но часто это оказывается удобным.

И наконец, начинается описание головной функции, для Windows-прило- жений она носит имя WinMain(). Рассмотрим ее заголовок:

int WINAPI WinMain(HINSTANCE This, // Дескриптор текущего приложения HINSTANCE Prev, // В современных системах всегда 0 LPSTR cmd, // Командная строка

int mode) // Режим отображения окна

Функция теперь имеет четыре параметра, устанавливаемых при загрузке про- граммы:

* This — дескриптор, устанавливаемый операционной системой для данного приложения;
* параметр Prev был предназначен для хранения дескриптора предыдущего экземпляра приложения, уже загруженного системой; сейчас потерял свою актуальность и сохранен лишь для совместимости со старыми приложе- ниями (начиная от Windows 95, он устанавливается в нулевое значение);
* cmd — обычная С-строка, где хранится командная строка, при помощи ко- торой запущено приложение, причем без имени запускаемой программы;
* mode — режим отображения окна.

Внутри головной функции описаны три переменные:

HWND hwnd; // Дескриптор главного окна программы MSG msg; // Структура для хранения сообщения

WNDCLASS wc; // Класс окна

* hwnd предназначен для хранения дескриптора главного окна программы. Идеология Windows предусматривает дескрипторы практически для всех объектов, причем их типы различаются, хотя, на самом деле, все дескрип- торы имеют тип int;
* msg — это структура, в которой будет храниться информация о сообще- нии, передаваемом операционной системой в приложение:

struct MSG

{

HWND hwnd; // Дескриптор окна UINT message; // Номер сообщения

WPARAM wParam; // 32-разрядные целые содержат LPARAM lParam; // дополнительные параметры сообщения

DWORD time; // Время посылки сообщения в миллисекундах POINT pt; // Координаты курсора (x,y)

};

struct POINT

{

LONG x,y;

};

* wc — структура, содержащая информацию по настройке окна. Требуется заполнить следующие поля:

 wc.hInstance = This;

Полю присваивается дескриптор текущего приложения.

 wc.lpszClassName = WinName;

Имя главного окна приложения совпадает с именем самого приложения.

 wc.lpfnWndProc = WindowsFunc;

Имя оконной функции для обработки сообщений.

 wc.style = 0;

Стиль окна. Нулевое значение задает стиль по умолчанию.

 wc.hIcon = LoadIcon(NULL,IDI\_APPLICATION);

Дескриптор иконки приложения. Функция LoadIcon() обеспечивает за- грузку иконки. Если первый параметр NULL, используется системная пиктограмма, которая выбирается заданием второго параметра:

* + IDI\_APPLICATION — стандартная иконка;
  + IDI\_ASTERISK — звездочки;
  + IDI\_EXCLAMATION — восклицательный знак;
  + IDI\_HAND — ладонь;
  + IDI\_QUESTION — вопросительный знак;
  + IDI\_WINLOGO — логотип Windows;

 wc.hCursor = LoadCursor(NULL,IDC\_ARROW);

Аналогично, функция LoadCursor() обеспечивает загрузку графическо- го образа курсора, где нулевой первый параметр также означает использование системного курсора, вид которого можно выбрать из списка:

* + IDC\_ARROW — стандартный курсор;
  + IDC\_APPSTARTING — стандартный курсор и маленькие песочные часы;
  + IDC\_CROSS — перекрестие;
  + IDC\_IBEAM — текстовый курсор;
  + IDC\_NO — перечеркнутый круг;
  + IDC\_SIZEALL — четырехлепестковая стрелка;
  + IDC\_SIZENESW — двухлепестковая стрелка, указывающая на северо- восток и юго-запад;
  + IDC\_SIZENWSE — двухлепестковая стрелка, указывающая на северо- запад и юго-восток;
  + IDC\_SIZENS — двухлепестковая стрелка, указывающая на север и юг;
  + IDC\_SIZEWE — двухлепестковая стрелка, указывающая на запад и восток;
  + IDC\_UPARROW — стрелка вверх;
  + IDC\_WAIT — песочные часы;

 wc.lpszMenuName = NULL;

Ссылка на строку главного меню, при его отсутствии — NULL.

 wc.cbClsExtra = 0;

Дополнительные параметры класса окна, если 0, параметры отсутст- вуют.

 wc.cbWndExtra = 0;

Аналогично, здесь указываются дополнительные параметры окна.

 wc.hbrBackground = (HBRUSH)GetStockObject(WHITE\_BRUSH);

Здесь задается дескриптор кисти, которая используется для заполнения окна. Можно использовать API-функцию GetStockObject(), которая создает стандартную кисть белого цвета WHITE\_BRUSH. Требуется явное преобразование типа — (HBRUSH).

После того, как определены основные характеристики окна, можно это окно создать при помощи API-функции CreateWindow(), для которой также нужно задать параметры:

1. WinName — имя, которое присвоено классу окна, поле lpszClassName.
2. "Каркас Windows-приложения" — С-строка, отображаемая в заголовке окна.
3. WS\_OVERLAPPEDWINDOW — макрос, определяющий стиль отображения стан- дартного окна, имеющего системное меню, заголовок, рамку для измене- ния размеров, а также кнопки минимизации, развертки и закрытия. Это наиболее общий стиль окна, он определен как:

#define WS\_OVERLAPPEDWINDOW (WS\_OVERLAPPED|WS\_CAPTION|WS\_SYSMENU| WS\_THICKFRAME|WS\_MINIMIZEBOX|WS\_MAXIMIZEBOX)

Можно создать другой стиль, используя комбинацию стилевых макросов при помощи операции логического сложения:

 WS\_OVERLAPPED — стандартное окно с рамкой.

 WS\_CAPTION — окно с заголовком.

 WS\_THICKFRAME — окно с рамкой.

 WS\_MAXIMIZEBOX — наличие кнопки развертки.

 WS\_MINIMIZEBOX — наличие кнопки минимизации.

 WS\_SYSMENU — наличие системного меню.

 WS\_HSCROLL — наличие горизонтальной панели прокрутки.

 WS\_VSCROLL — наличие вертикальной панели прокрутки.

1. Следующие два параметра определяют координаты левого верхнего угла окна (x,y), еще два параметра: Width — ширину и Height — высоту окна

в пикселах. Задание параметра CW\_USEDEFAULT означает, что система сама выберет для отображения окна наиболее (с ее точки зрения) удобное место и размер.

1. Следующий параметр — указатель на структуру меню, или NULL, при его отсутствии.
2. Далее требуется указать дескриптор приложения владельца окна — This.
3. И, наконец, указатель на дополнительную информацию, у нас — NULL.

Окно создано, и с ним можно работать, но окно пока не отображается. Для того чтобы его увидеть, необходимо окно показать функцией ShowWindow(hwnd, mode), которая принимает два параметра: hwnd — дескрип- тор окна и mode — режим отображения, мы используем значение, полученное при открытии приложения через параметр головной функции WinMain().

Далее, заключительная часть головной функции — цикл обработки сообще- ний. Он задается оператором while, аргументом которого является функция GetMessage(&msg, NULL, 0, 0). Такой цикл обязателен для всех Windows- приложений и используется для получения и обработки сообщений, переда- ваемых операционной системой. Сообщения ставятся в очередь, откуда и из- влекаются, по мере готовности приложения, функцией GetMessage():

* первым параметром функции является &msg — указатель на структуру MSG, где и сохраняются сообщения;
* второй параметр hwnd — определяет окно, для которого предназначено сообщение, если же необходимо перехватить все сообщения данного при- ложения, то он должен быть NULL;
* остальные два параметра определяют [min, max] диапазон получаемых сообщений. Чаще всего необходимо обработать все сообщения, тогда эти параметры должны быть равны 0.

Если очередь сообщений пуста, функция возвращает 0 и управление переда- ется операционной системе.

Внутри цикла расположены две функции:

TranslateMessage(&msg); DispatchMessage(&msg);

Первая из них транслирует код нажатой клавиши в клавиатурные сообщения WM\_CHAR. При этом в переменную wParam структуры msg помещается ASCII-код нажатой клавиши, а в lParam битовая карта со значениями, приведенными в табл. 5.2.

Использование этой функции необязательно и нужно только для обработки кода нажатой клавиши.

***Таблица 5.2.*** *Битовая карта клавиатуры*

|  |  |
| --- | --- |
| **Бит** | **Значение** |
| 15 | 1, если клавиша отпущена, 0 — если нажата |
| 14 | 1, если клавиша была нажата перед посылкой сообщения |
| 13 | 1, если нажата клавиша <Alt> |
| 12-9 | Системные |
| 8 | 1, если нажата функциональная клавиша |
| 7-0 | Scan-код клавиши |

Вторая функция DispatchMessage(&msg) обеспечивает возврат преобразован- ного сообщения обратно операционной системе и инициирует вызов оконной функции данного приложения для его обработки.

Данным циклом и заканчивается функция WinMain().

Нам осталось лишь описать оконную функцию, и построение каркаса Windows-приложения будет закончено.

Основной и единственный компонент этой функции — переключатель, обес- печивающий выбор соответствующего обработчика сообщений по его номе- ру message. Здесь мы предусмотрели обработку лишь одного сообщения WM\_DESTROY. Это сообщение посылается, когда пользователь завершает про- грамму. Получив это сообщение, оконная функция вызывает API-функцию PostQuitMessage(0), которая завершает приложение и передает операционной системе код возврата 0. Если говорить точнее, генерируется сообщение WM\_QUIT, получив которое функция GetMessage() возвращает нулевое значе- ние, в результате цикл обработки сообщений прекращается и происходит за- вершение работы приложения.

Все остальные сообщения обрабатываются "по умолчанию" функцией DefWindowProc(), имеющей такой же список параметров и аналогичное воз- вращаемое значение, поэтому вызов функции и помещается как возвращае- мое значение оператора return.

На этом описание каркаса Windows-приложения завершено.

## Обработка сообщений

Операционная система способна генерировать больше двух сотен сообщений. В файле включений winuser.h размещены макроимена для них. Этот файл не нужно подключать самостоятельно, поскольку он вызывается неявно через

windows.h. Рассмотрим технику обработки наиболее распространенных сооб- щений Windows.

### Нажатие клавиши

При нажатии любой клавиши на клавиатуре вырабатывается сообщение WM\_CHAR. Чтобы обработать это сообщение, необходимо добавить в переклю- чатель оконной функции еще одну строку альтернативы:

case WM\_CHAR :

и описать необходимые действия для обработки нажатия клавиши.

Поставим самую простую задачу — при нажатии на клавишу выводить теку- щий символ в окне приложения, т. е. обеспечить эхо-печать в одну строку, пока не беспокоясь о выходе строки за границу окна.

Для решения этой задачи нам понадобится на глобальном уровне описать текстовый массив для хранения набранных символов.

char str[80] = "";

Глобальный уровень нам понадобился по той причине, что содержимое тек- стовой строки должно сохранять свое значение и при выходе из оконной функции.

Следующая проблема, которую нужно решить, — это организация вывода строки в окно приложения. Для этого необходим *контекст устройства*. В Windows все функции, выводящие что-либо в окно, используют в качестве параметра *дескриптор* контекста устройства, который представляет собой структуру, описывающую свойства данного устройства вывода. А дескрип- тор — это индекс, который позволяет операционной системе извлечь нужный контекст из массива аналогичных структур. Для получения контекста устрой- ства необходимо в оконной функции описать переменную — *дескриптор контекста устройства*:

HDC hdc;

а в обработчике сообщений получить этот дескриптор вызовом функции

GetDC():

hdc = GetDC(hwnd);

Теперь можно выводить текст в окно с помощью функции TextOut(): BOOL WINAPI TextOut(HDC hdc, int x, int y, LPCSTR str, int len);

которая принимает в качестве параметров контекст устройства hdc, (x,y) — координаты начала вывода текста, указатель на С-строку с текстом и длину выводимой строки.

Необходимо также учитывать, что определение контекста устройства приво- дит к выделению области памяти для записи структуры, содержащей инфор- мацию о данном устройстве вывода. И после того как строка будет выведена и необходимость в данном контексте отпадет, его нужно уничтожить, т. е. освободить память. Дело в том, что даже после выхода из оконной функции, блок памяти под контекстом устройства автоматически не освобождается, и если его не освободить принудительно, то память будет непроизводительно расходоваться. Для освобождения контекста устройства вызывается функция:

ReleaseDC(hwnd,hdc);

где в качестве параметров фигурируют дескрипторы окна и контекста устройства.

Листинг 5.2 представляет собой текст переработанной нами части про- граммы.

**Листинг 5.2**

char str[80] = "";

int strApp(char ch) // Функция добавления символа к строке

{

int i = strlen(str); str[i] = ch; str[++i] = '\0'; return i;

}

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

{

HDC hdc; switch(message)

{ // Обработка нажатия клавиши case WM\_CHAR : hdc = GetDC(hwnd);

TextOut(hdc,1,1,str,strApp((char)wParam)); ReleaseDC(hwnd,hdc);

break;

case WM\_DESTROY : PostQuitMessage(0); break;

default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}

Теперь продолжим обсуждение текста программы. Функция в качестве пара- метра, кроме задания строки текста, требует еще и длину этой строки. Мы

решили сразу "убить двух зайцев" и написали функцию strApp(), которая до- бавляет символ к строке и возвращает ее длину.

Для вычисления длины существующей строки воспользуемся готовой функ- цией:

int i = strlen(str);

добавлять файл включений string.h нет необходимости, поскольку он под- ключается неявно. Далее все просто, добавим принятый с клавиатуры символ и завершающий символ строки:

str[i] = ch; str[++i] = '\0';

теперь осталось только вернуть индекс i, который равен длине полученной строки

return i;

и функция завершена. Обращение к этой функции:

strApp((char)wParam);

Что выглядит довольно интересно и странно, но все становится ясно, если вспомнить, что переменная wParam возвращает ASCII-код символа, а операция явного преобразования типа (char)wParam просто выделит этот символ и пе- редаст его как параметр функции. При этом, поскольку управление функции TextOut() будет передано лишь после того, как будут определены все пара- метры, строка уже станет больше на принятый с клавиатуры символ.

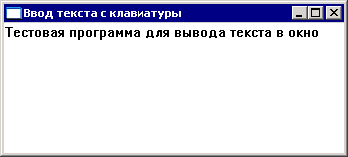
Теперь следует обсудить позицию, куда будет выведена текстовая строка. Система координат (рис. 5.5), используемая при выводе в окно, имеет начало в верхнем левом углу окна, ось x направлена горизонтально, а ось y — верти- кально вниз. Причем вывод осуществляется в клиентскую область окна, т. е. исключая полосу заголовка.



**Рис. 5.5.** Система координат окна

По умолчанию размер окна измеряется в пикселах, т. е. точках на экране. Та- ким образом, вывод в точку с координатами (1, 1) — это практически с нача- ла окна.

Запустим программу на выполнение и введем текст, должно получиться что- то похожее на рис. 5.6.



**Рис. 5.6.** Вид окна программы эхо-печати

Текст выводится шрифтом по умолчанию, и пока нас это устраивает, а как установить необходимый шрифт, мы обсудим позднее. Также мы не стали выводить текст по одному символу, а каждый раз просто выводим вновь с тех же координат всю строку целиком, и в таком подходе есть некоторый смысл.

Однако если свернуть окно и развернуть его вновь, мы увидим, что выведен- ная в окне строка исчезла. При изменении размера окна мы также можем по- терять часть строки. Дело в том, что при необходимости обновления окна операционная система генерирует сообщение WM\_PAINT, которое в нашем слу- чае обработано по умолчанию.

Для того чтобы избежать этой неприятной ситуации, построим обработчик сообщения WM\_PAINT для вывода строки текста при перерисовке окна. Текст обработчика сообщения выглядит довольно просто:

case WM\_PAINT : hdc = BeginPaint(hwnd,&ps);

TextOut(hdc,1,1,str,strlen(str)); EndPaint(hwnd,&ps);

break;

Однако здесь имеются некоторые особенности, в частности, дескриптор кон- текста устройства должен возвращаться функцией BeginPaint(), принимаю- щей два параметра:

* hwnd — дескриптор окна.
* &ps — указатель на структуру PAINTSTRUCT.

Освобождаться контекст устройства должен функцией EndPaint() с теми же параметрами. Переменная типа PAINTSTRUCT предварительно должна быть описана в оконной функции:

PAINTSTRUCT ps;

Собственно, особенность описания обработчика сообщения WM\_PAINT связана с возможностью перерисовки только части окна, а необходимая для этого информация содержится в структуре PAINTSTRUCT:

struct PAINTSTRUCT {

|  |  |  |
| --- | --- | --- |
| HDC | hdc; | // Дескриптор контекста устройства |
| BOOL | fErase; |  |
| RECT | rcPaint; | // Координаты прямоугольной области перерисовки |
| BOOL | fRestore; |  |
| BOOL | fIncUpdate; |  |

BYTE rgbReserved[32];

};

Если поля структуры ps не заданы, то перерисовывается все окно.

Теперь можно безболезненно изменять размеры окна, не беспокоясь о со- хранности текста в окне. Однако текст, выходящий за границу окна, отобра- жаться не будет.

### Сообщение мыши

Нажатие на клавиши мыши приводит к следующим сообщениям:

* WM\_LBUTTONDOWN — нажатие на левую кнопку мыши;
* WM\_RBUTTONDOWN — нажатие на правую кнопку мыши.

Составим тестовую программу для демонстрации обработки этих сообщений, просто добавив к оконной функции Windows-приложения два обработчика сообщений (листинг 5.3).



Поскольку каркас приложения изменяется лишь в части касающейся обработки сообщений, то мы не будем приводить полного листинга, а только изменяемую часть — обработчики сообщений, а при необходимости — всю оконную функ- цию. В головной же функции будем изменять лишь строку заголовка окна.

|  |
| --- |
| **Листинг 5.3. Обработчики сообщений нажатия на кнопку мыши** |

case WM\_LBUTTONDOWN : strcpy(str,"Нажата левая кнопка мыши"); x = LOWORD(lParam);

y = HIWORD(lParam); InvalidateRect(hwnd,NULL,1); break;

case WM\_RBUTTONDOWN : strcpy(str," Нажата правая кнопка мыши "); x = LOWORD(lParam);

y = HIWORD(lParam); InvalidateRect(hwnd,NULL,1); break;

Здесь мы реализуем стандартный подход к организации вывода данных в ок- но. Обычно весь вывод организуют в обработчике сообщения WM\_PAINT, что позволяет просто организовать перерисовку окна в случае необходимости. Обработчики же остальных сообщений должны подготовить информацию для вывода в окно и инициировать принудительно перерисовку окна.

Так мы и поступим, скопируем соответствующий текст в строку str глобаль- ной области видимости и вернем в переменные x, y — координаты курсора. Эти координаты нам понадобятся для вывода строки текста с этой позиции, а извлечем мы их из параметра lParam, где младшее слово содержит x-коор- динату, а старшее слово — y-координату. Для этого используем макросы, описанные в файле включений windef.h:

x = LOWORD(lParam); y = HIWORD(lParam);

Эти макросы можно заменить простыми конструкциями:

x = lParam & 0xFFFF;

y = (lParam >> 16) & 0xFFFF;

Теперь необходимо спровоцировать систему на перерисовку окна, не можем же мы из одного обработчика сообщения вызывать другой. Однако это мож- но сделать, обратившись к функции:

InvalidateRect(HWND hwnd, RECT \*rect, BOOL fErase);

Эта функция объявляет область окна hwnd, заданную вторым параметром rect, *недействительной*, т. е. подлежащей перерисовке, что заставляет Windows генерировать сообщение WM\_PAINT. Последний параметр fErase — флаг перерисовки заполнения окна, если значение fErase отлично от нуля, то вначале изображение полностью стирается и рисование начинается с запол- нения фона, иначе фоновое заполнение не перерисовывается.

Нетрудно сообразить, что параметры этой функции используются для запол- нения структуры PAINTSTRUCT.

В нашем случае обращение к функции выглядит так:

InvalidateRect(hwnd,NULL,1);

где вторым параметром указано NULL, что означает необходимость перери- совки всего окна.

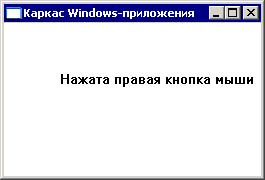
Теперь нам необходимо определить обработчик сообщения WM\_PAINT: case WM\_PAINT : hdc = BeginPaint(hwnd,&ps);

TextOut(hdc,x,y,str,strlen(str)); EndPaint(hwnd,&ps);

break;

который отличается от приведенного в предыдущем примере лишь тем, что функция TextOut() выводит с точки (x,y).

Запустив программу на выполнение, получим картинку, изображенную на рис. 5.7.



**Рис. 5.7.** Обработка нажатия кнопки мыши

### Создание окна

При создании окна генерируется сообщение WM\_CREATE, которое происходит еще до отображения окна и позволяет производить некоторые начальные установки. Мы продемонстрируем использование этого сообщения в сле- дующем примере.

### Таймер

В программе можно установить один или несколько *таймеров*, которые по- зволяют производить отсчет времени. Таймер создается функцией SetTimer():

UINT SetTimer(HWND hwnd, UINT nID, UINT wLength, TIMEPROC lpTFunc);

* hwnd — дескриптор окна;
* nID — номер таймера, задается целым числом;
* wLength — временной интервал таймера в миллисекундах;
* lpTFunc — указатель на функцию, вызываемую при обработке таймера. Эта функция должна быть определена как VOID CALLBACK и имеет парамет- ры, показанные ниже в прототипе, но, если lpTFunc = NULL, для обработки

сообщения таймера вызывается оконная функция. Обычно так и посту- пают.

VOID CALLBACK TimerProc(

HWND hwnd, // Дескриптор окна UINT uMsg, // WM\_TIMER сообщение

INT idEvent, // Номер таймера

DWORD dwTime // Текущее системное время

);

Если поставить задачу отсчета времени с момента начала работы приложе- ния, то проще всего создать таймер по сообщению WM\_CREATE:

case WM\_CREATE : t = 0;

SetTimer(hwnd,1,1000,NULL); break;

Здесь мы присвоим начальное значение переменной t = 0, которую опишем на глобальном уровне и будем использовать как счетчик секунд. Таймеру присвоим номер 1, а интервал отсчета зададим в 1000 миллисекунд (1 секун- да), в качестве же обработчика таймера используем оконную функцию (тре- тий параметр NULL).

Теперь опишем обработчик таймера:

case WM\_TIMER : t++;

strcpy(str,"Секунды: ");

strcat(str,itoa(t,s,10)); InvalidateRect(hwnd,NULL,1); break;

Здесь мы сформируем строку текста для вывода в окне, используя функцию itoa() для преобразования целого числа в строку, где третий параметр — основание системы счисления 10. После чего инициируем вызов сообщения WM\_PAINT.

Пример текста оконной функции для таймера приведен в листинге 5.4.

**Листинг 5.4. Оконная функция для таймера**

char s[10], str[20]; int t;

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message,

WPARAM wParam, LPARAM lParam)

{

HDC hdc; PAINTSTRUCT ps;

// Обработчик сообщений

switch(message)

{

case WM\_CREATE : t = 0;

SetTimer(hwnd,1,1000,NULL); break;

case WM\_TIMER : t++;

strcpy(str,"Секунды: ");

strcat(str,itoa(t,s,10)); InvalidateRect(hwnd,NULL,1); break;

case WM\_PAINT : // Перерисовка экрана hdc = BeginPaint(hwnd,&ps); TextOut(hdc,1,1,str,strlen(str)); EndPaint(hwnd,&ps);

break;

case WM\_DESTROY : PostQuitMessage(0); break;// Завершение программы

// Обработка сообщения по умолчанию

default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}

На глобальном уровне описаны два символьных массива s, str и переменная целого типа t — счетчик секунд. И если массив s можно было бы описать внутри оконной функции, поскольку он используется локально лишь в одном обработчике сообщений, то остальные переменные должны быть описаны именно на глобальном уровне, т. к. их значение устанавливается в одном об- работчике, а используются они в другом. Нужно помнить, что переход от од- ного обработчика к другому происходит лишь после передачи управления операционной системе и получения следующего сообщения, а локальная пе- ременная в этом случае потеряет свое значение.



Можно, конечно, описывать переменные как статические и в оконной функции:

static char s[10], str[20]; static int t;

в этом случае их значение сохранится до следующего вызова.

Для уничтожения таймера используется функция KillTimer(), которая воз- вращает true при успешном удалении таймера.

BOOL WINAPI KillTimer(HWND hwnd, UINT nID);

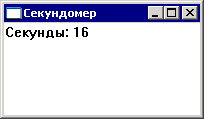
Параметры имеют тот же смысл, что и для функции SetTimer().

В нашем случае было бы корректным вызвать эту функцию при завершении работы в сообщении WM\_DESTROY:

case WM\_DESTROY: KillTimer(hwnd,1);

...

Пример работы программы показан на рис. 5.8.



**Рис. 5.8.** Таймер

### Рисование в окне

При рисовании в окне можно использовать достаточно большой набор API- функций, для которых нужно учитывать следующее соглашение — все линии рисуются текущим пером, а области заполняются текущей кистью. Как уста- навливать перо и кисть мы обсудим позднее, а пока будем рисовать пером и кистью, определенными при создании окна. Все эти функции возвращают ненулевое значение в случае успешного выполнения, и 0 — в случае ошибки. Начнем с рассмотрения с самых простых функций.

#### Рисование линии

Для того чтобы нарисовать линию, используется функция:

BOOL LineTo(HDC hdc, int x, int y);

где hdc — дескриптор контекста устройства, x, y — координаты конца ли- нии. Начало линии определяется положением текущей позиции рисования.

При инициализации графической системы функцией начальная позиция оп- ределяется в начале координат (0,0). Если же необходимо нарисовать линию из другой точки, то текущая позиция рисования может быть изменена.

#### Установка текущей позиции

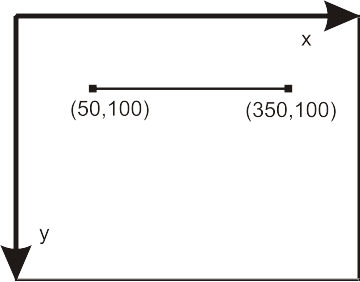
BOOL MoveToEx(HDC hdc, int x, int y, LPPOINT oldPoint);

Функция устанавливает текущую позицию в точку с координатами (x,y) и передает в структуру POINT координаты предыдущей позиции. Если послед- ний параметр NULL, предыдущие координаты не сохраняются.

Обычно эти функции используются в паре, обе они возвращают ненулевое значение в случае успешного завершения, и 0 — в случае ошибки.

Например, если нам нужно нарисовать линию между двумя точками (рис. 5.9), можно сделать это так:

MoveToEx(hdc,50,100,NULL); LineTo(hdc,350,100);



**Рис. 5.9.** Рисование линии

Функции рисования можно включать в любой обработчик сообщений, но по многим соображениям лучше всего — в сообщение WM\_PAINT. Хотя бы пото- му, что операционная система автоматически генерирует это сообщение при необходимости перерисовки окна, а значит, если рисовать в окне посредст- вом других обработчиков, то мы потеряем выведенную графику при перери- совке окна.

#### Определение размера клиентской области

При выводе данных в окно необходимо определить фактический размер этого окна, точнее его клиентской области, т. е. окна без верхней строки заголовка. Для решения данной задачи существует API-функция:

BOOL WINAPI GetClientRect(HWND hWnd, LPRECT lpRect);

принимающая дескриптор окна hWnd и адрес структуры RECT — lpRect, опре- деляющей его размеры:

struct RECT

{

|  |  |  |
| --- | --- | --- |
| LONG  LONG | left;  top; | // x – координата левого верхнего угла  // y - координата левого верхнего угла |
| LONG | right; | // Ширина |
| LONG | bottom; | // Высота |

};

В результате работы функции будут заполнены поля структуры RECT, и мы можем использовать высоту и ширину окна в последующих графических по- строениях.



В данном контексте применения функции GetClientRect(), поля left и top

структуры RECT будут равны 0.

Теперь у нас достаточно информации, чтобы рассмотреть простой пример. Поставим задачу построить оси системы координат, проходящие через центр окна. Причем картинка должна перестраиваться при изменении размеров ок- на. В листинге 5.5 приведен текст оконной функции.

**Листинг 5.5**

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

{

HDC hdc; PAINTSTRUCT ps; RECT rt;

switch(message) {

case WM\_PAINT : hdc = BeginPaint(hwnd,&ps); GetClientRect(hwnd, &rt); MoveToEx(hdc,0,rt.bottom/2,NULL); LineTo(hdc,rt.right,rt.bottom/2); MoveToEx(hdc,rt.right/2,0,NULL); LineTo(hdc,rt.right/2,rt.bottom); EndPaint(hwnd,&ps);

break;

case WM\_SIZE :InvalidateRect(hwnd,NULL,1); break; case WM\_DESTROY : PostQuitMessage(0); break;

default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}

#### Изменение размеров окна

При изменении размеров окна система генерирует сообщение WM\_SIZE, реаги- руя на которое мы должны перерисовать окно, поскольку его размеры изме- нились, и линии осей системы координат должны быть перестроены. Как и ранее, мы решим эту задачу, просто вызывая функцию InvalidateRect(), ко-

торая и обеспечит генерацию сообщения WM\_PAINT, что приведет к перерисов- ке окна. Обработчик сообщения будет следующим:

case WM\_SIZE :InvalidateRect(hwnd,NULL,1); break;

Теперь мы можем изменять размер окна и система координат всегда будет подстраиваться под текущий размер.

#### Рисование прямоугольника

Нарисовать прямоугольник можно при помощи функции:

BOOL Rectangle(HDC hdc, int x1, int y1, int x2, int y2);

где (x1,y1) — координаты левого верхнего угла прямоугольника, а (x2,y2) — координаты правого нижнего угла. Прямоугольник заполняется текущей кистью, поэтому если к предыдущему примеру (листинг 5.5) доба- вить в обработчик сообщения WM\_PAINT следующую строку:

Rectangle(hdc,rt.right/4,rt.bottom/4,3\*rt.right/4,3\*rt.bottom/4);

которая рисует прямоугольник в четверть окна, то мы получим картинку, изображенную на рис. 5.10.



**Рис. 5.10.** Окно программы рисования системы координат

#### Рисование эллипса

Функция для отображения эллипса имеет те же параметры, поскольку эллипс определяется ограничивающим его прямоугольником:

BOOL Ellipse (HDC hdc, int x1, int y1, int x2, int y2);

а внутренняя область также заполняется текущей кистью. Если необходимо нарисовать круг, нужно задать равносторонний прямоугольник.

Таким образом, если мы заменим Rectangle() на Ellipse(), то получим вме- сто прямоугольника эллипс.

Ellipse(hdc,rt.right/4,rt.bottom/4,3\*rt.right/4,3\*rt.bottom/4);

#### Рисование точки

Функция SetPixel() позволяет вывести в окно одну точку (пиксел):

COLORREF SetPixel(HDC hdc, int x, int y, COLORREF color);

где hdc — контекст устройства, (x,y) — координаты точки, color — цвет точки. Функция возвращает прежний цвет точки в случае успешного завер- шения, и –1 — при ошибке.

Тип COLORREF представляет длинное целое число, где три младших байта ко- дируют соответственно *красный*, *синий*, *зеленые* цвета. Старший байт всегда

1. Числовое значение байта определяет интенсивность каждого цвета, а цвет точки определяется смешиванием трех базовых цветов. Для получения нуж- ного цвета проще всего воспользоваться предопределенным в файле wingdi.h макросом RGB(), прототип которого можно представить в виде:

COLORREF RGB(int red, int green, int blue);

Например, если нам нужен чисто зеленый цвет, мы могли бы определить пе- ременную color так:

COLORREF color = RGB(0,255,0);

Для демонстрации рассмотренных функций рисования напишем небольшую программу, точнее ее оконную функцию (листинг 5.6), которая строит про- стой график.

**Листинг 5.6. Построение синусоиды**

const double PI = 3.14159;

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

{

HDC hdc; PAINTSTRUCT ps; RECT rt;

COLORREF color = RGB(0,255,0); // Цвет зеленый int a,b,x\_scr,y\_scr,h\_scr; // Экранные координаты double x, y, h\_x; // Физические координаты switch(message)

{

case WM\_PAINT : hdc = BeginPaint(hwnd,&ps); GetClientRect(hwnd, &rt);

a = rt.right/2; // Половина ширины окна b = rt.bottom/2; // Половина высоты окна MoveToEx(hdc,0,b,NULL);

LineTo(hdc,2\*a,b);

MoveToEx(hdc,a,0,NULL); LineTo(hdc,a,2\*b); h\_scr = 1;

h\_x = PI\*h\_scr/a;

for (x=-PI,x\_scr=0; x<PI; x+=h\_x,x\_scr+=h\_scr)

{

y = sin(x); y\_scr = b - y\*b;

SetPixel(hdc, x\_scr, y\_scr, color);

}

EndPaint(hwnd,&ps); break;

case WM\_SIZE :InvalidateRect(hwnd,NULL,1); break;

case WM\_DESTROY : PostQuitMessage(0); break; // Завершение программы

// Обработка сообщения по умолчанию

default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}

Попробуем прокомментировать текст данной программы. В качестве основы для ее создания мы взяли предыдущую программу с уже построенной систе- мой координат, где для упрощения формул ввели две локальные переменные: a, b, равные, соответственно, половине ширины и высоты окна. После вывода осей системы координат можно начинать строить кривую, но вначале необ- ходимо определить как физические координаты, присутствующие в формуле, и экранные координаты будут связаны между собой. То, что для вывода в окно нельзя использовать физические координаты, должно быть ясно. Дейст- вительно, если наша функция имеет значение в диапазоне (–1, 1), то при ото- бражении такой картинки в окне мы ничего не сможем увидеть.

Таким образом, задача заключается в том, чтобы растянуть картинку как по горизонтали, так и по вертикали. И если физическая координата в начале по- строения равна x=-PI, то экранная координата должна равняться x\_scr = 0. Здесь мы на глобальном уровне определили константу:

const double PI = 3.14159;

В конце построения должно получиться x = PI, x\_scr = 2\*a.

Чтобы достичь такого результата, зададим шаг изменения экранной х-коор- динаты h\_scr=1, а шаг изменения физической координаты вычислим по фор- муле:

h\_x = PI\*h\_scr/a;

Теперь осталось организовать цикл, изменяя одновременно физическую и экранную координаты на величину заданного шага. Внутри цикла вычисляем значение функции (не забыть добавить файл включений math.h для функции sin(x)), затем вычисляем экранную y\_scr-координату, учитывая, что направ- ление физической и экранной оси y не совпадает, а начало физических коор- динат находится в середине окна. Отсюда и формула:

y\_scr = b - y\*b;

После чего строим точку в окне:

SetPixel(hdc, x\_scr, y\_scr, color);

Зеленый цвет точки мы задали при описании переменной color: COLORREF color = RGB(0,255,0);

Завершаем обработчик сообщения функцией EndPaint(hwnd,&ps), после чего можно программу выполнить.

Если все введено правильно, то мы увидим в окне синусоиду, которая пере- страивается при каждом изменении размеров окна, сохраняя свое положение относительно заданной системы координат.

Однако, анализируя полученный график, мы видим, что на нем вместо непре- рывной линии получился набор точек. Эту проблему можно решить, если вместо установки точки мы рисовали бы отрезок ломаной линии функцией LineTo(), но тогда мы получим линию черного цвета, поскольку линия рису- ется текущим пером.

#### Создание пера

Создать *перо* можно при помощи функции:

HPEN CreatePen(int style, int width, COLORREF color);

Параметр style определяется макросом и задает тип линии, список его воз- можных значений приведен в табл. 5.3.

***Таблица 5.3.*** *Макросы, определяющие тип линии*

|  |  |
| --- | --- |
| **Макрос** | **Тип линии** |
| PS\_DASH | Пунктирная линия (---) |
| PS\_DASHDOT | Штрихпунктирная линия (-.-) |
| PS\_DASHDOTDOT | Штрихпунктирная линия (-..-) |
| PS\_DOT | Точечная линия (...) |
| PS\_INSIDEFRAME | Сплошная линия внутри области |

***Таблица 5.3*** *(окончание)*

|  |  |
| --- | --- |
| **Макрос** | **Тип линии** |
| PS\_NULL | Прозрачное перо |
| PS\_SOLID | Сплошная линия |

Параметр width задает ширину пера в логических единицах. Если ширина пера установлена в 0, то линии рисуются в 1 пиксел независимо от режима рисования.

По умолчанию одна логическая единица равна одному пикселу.  Пунктирные линии рисуются только при ширине пера в 1 пиксел.



И наконец, параметр color задает цвет пера. Как задать цвет, мы уже рас- сматривали, но далее, в табл. 5.4, приведем список чистых цветов. Дело в том, что перу всегда присваивается *чистый цвет*, и если заданный цвет не является чистым, то линии будут рисоваться ближайшим чистым цветом, за исключением стиля PS\_INSIDEFRAME и при ширине линии более 1 пиксела.

***Таблица 5.4.*** *Значения параметров макроса RGB для генерации чистых цветов*

|  |  |
| --- | --- |
| **Цвет** | **Red, Green, Blue** |
| Темно-красный | 128, 0, 0 |
| Светло-красный | 255, 0, 0 |
| Темно-зеленый | 0, 128, 0 |
| Светло-зеленый | 0, 255, 0 |
| Темно-синий | 0, 0, 128 |
| Светло-синий | 0, 0, 255 |
| Темно-желтый | 128, 128, 0 |
| Светло-желтый | 255, 255, 0 |
| Темно-бирюзовый | 0, 128, 128 |
| Светло-бирюзовый | 0, 255, 255 |
| Темно-сиреневый | 128, 0, 128 |
| Светло-сиреневый | 255, 0, 255 |
| Черный | 0, 0, 0 |
| Темно-серый | 128, 128, 128 |

***Таблица 5.4*** *(окончание)*

|  |  |
| --- | --- |
| **Цвет** | **Red, Green, Blue** |
| Светло-серый | 192, 192, 192 |
| Белый | 255, 255, 255 |

После создания перо выбирается в контексте устройства при помощи функции:

HGDIOBJ SelectObject(HDC hdc, HGDIOBJ hpen);

где указываются контекст устройства вывода и дескриптор созданного функ- цией CreatePen() пера. В качестве примера установим синее перо толщиной два пиксела в программе построения системы координат:

HPEN hpen;

hpen = CreatePen(PS\_SOLID,2,RGB(0, 0, 255)); SelectObject(hdc,hpen);

Нетрудно убедиться, что линии действительно стали синими и в два раза толще.

Можно и не создавать перо, а воспользоваться готовыми перьями Windows. В этом случае дескриптор пера можно получить при помощи функции GetStockObject():

HGDIOBJ GetStockObject(int);

указывая в качестве параметра соответствующий макрос:

* BLACK\_PEN — черное перо;
* WHITE\_PEN — белое перо;
* NULL\_PEN — прозрачное перо.



Поскольку функция может возвращать различные объекты, то для возвращае- мого значения необходимо указывать явно преобразование типа, например:

HPEN hpen = (HPEN)GetStockObject(BLACK\_PEN);

Перед выходом из программы все созданные перья необходимо удалить (за исключением системных), поскольку каждое вновь созданное перо занимает определенные ресурсы, которые автоматически не освобождаются при пере- ходе на другое перо. Сделать это нужно функцией DeleteObject():

BOOL DeleteObject(HGDIOBJ hpen);

Переделаем немного программу вывода графика синусоиды с учетом воз- можности создавать и менять перья в процессе вывода. Будем выводить оси координат синей линией толщиной 2 пиксела, а кривую — сплошной красной линией толщиной также 2 пиксела, в этом случае мы можем выбрать шаг цикла по оси х в 3 пиксела. Здесь мы воспользуемся тем обстоятельством, что функция LineTo() перемещает текущую позицию в конец линии. Определе- ние перьев вынесем отдельно в сообщение WM\_CREATE, а их удаление — в WM\_DESTROY. В листинге 5.7 приведен текст преобразованной программы.

**Листинг 5.7**

const double PI = 3.14159; HPEN hpen1, hpen2;

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

{

HDC hdc; PAINTSTRUCT ps; RECT rt;

int a, b, x\_scr, y\_scr, h\_scr; double x, y, h\_x; switch(message)

{

case WM\_CREATE :hpen1 = CreatePen(PS\_SOLID,2,RGB(0, 0, 255)); hpen2 = CreatePen(PS\_SOLID,2,RGB(255, 0, 0));

break;

case WM\_PAINT : hdc = BeginPaint(hwnd,&ps); GetClientRect(hwnd, &rt);

a = rt.right/2; b = rt.bottom/2;

SelectObject(hdc,hpen1); // Синее перо MoveToEx(hdc,0,b,NULL); LineTo(hdc,2\*a,b); MoveToEx(hdc,a,0,NULL); LineTo(hdc,a,2\*b);

h\_scr = 3;

h\_x = PI\*h\_scr/a;

SelectObject(hdc,hpen2); // Красное перо MoveToEx(hdc,0,b,NULL);

for (x=-PI,x\_scr=0; x<PI; x+=h\_x,x\_scr+=h\_scr)

{

y = sin(x); y\_scr = b - y\*b;

LineTo(hdc, x\_scr, y\_scr);

}

EndPaint(hwnd,&ps); break;

case WM\_SIZE :InvalidateRect(hwnd,NULL,1); break;

case WM\_DESTROY:DeleteObject(hpen1); DeleteObject(hpen2); PostQuitMessage(0);

break;

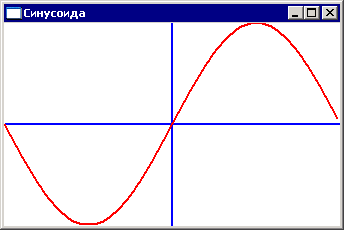
default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}

В результате работы программы получаем более привлекательный график, который приведен на рис. 5.11.



**Рис. 5.11.** Вывод графика функции сплошной линией

#### Настройка графического режима

Как показал предыдущий пример, построение графика функций происходит довольно неуклюже. Действительно, нам всегда нужно помнить, что система координат имеет начало в левом верхнем углу окна, а ось y направлена вниз. Поскольку мы привыкли строить графики в декартовой системе координат, нам приходится каждый раз пересчитывать физические координаты в коор- динаты экранные. Все это можно делать, но система программирования пре- доставляет набор функций, которые позволят упростить процесс и устано- вить такую систему координат, которая необходима при решении задачи. Можно задать размер выводимой области, его физические размеры, а также

новое начало координат и направление координатных осей. Но вначале рас- смотрим возможные режимы отображения графики.

##### Режимы отображения

По умолчанию устанавливается графический режим MM\_TEXT, при котором одна логическая единица равна 1 экранному пикселу. Этот режим обычно применяется для вывода текста, но при выводе графики обычно используют другие режимы.



Установка режима отображения не меняет ни размера окна, ни разрешающей способности, а лишь изменяет способ преобразования логических координат в экранные пикселы.

Для установки текущего режима отображения используется функция

SetMapMode():

int SetMapMode(HDC hdc, int mode);

где параметр mode определяет графический режим и принимает одно из за- данных значений:

* MM\_TEXT — одна логическая единица равна 1 пикселу (этот режим выбран по умолчанию);
* MM\_LOMETRIC — одна логическая единица равна 0,1 миллиметра;
* MM\_HIMETRIC — одна логическая единица равна 0,01 миллиметра;
* MM\_LOENGLISH — одна логическая единица равна 0,01 дюйма;
* MM\_HIENGLISH — одна логическая единица равна 0,001 дюйма;
* MM\_TWIPS — одна логическая единица равна 1/12 точки принтера или 1/1440 дюйма;
* MM\_ISOTROPIC — режим отображения логических единиц с эквивалентным масштабированием по осям координат.
* MM\_ANISOTROPIC — режим отображения логических единиц с различным масштабированием по осям координат.

##### Определение логических размеров окна

При использовании режимов отображения MM\_ISOTROPIC и MM\_ANISOTROPIC не- обходимо задать размеры окна в логических координатах. Здесь нужно учи- тывать то обстоятельство, что с этими логическими координатами мы будем обращаться к графическим функциям, где они интерпретируются как имею- щие тип int, в связи с чем логические координаты часто не совпадают с фи-

зическими координатами. Для примера вернемся к предыдущей задаче, гра- фик которой приведен на рис. 5.11. У рассматриваемой функции sin(x) аргу- мент x меняется в диапазоне (- ; ), а сама функция принимает значения в диапазоне (–1; 1). Если бы мы задали логический размер окна (6,28; 2), то вряд ли мы получили бы в окне вывода что-либо разумное. Поэтому мы вве- дем масштабный множитель и зададим размер так: (628; 400), т. е. умножим x-координату на 100, а y-координату — на 200.





При задании логических размеров окна лучше задавать их как можно ближе к соотношению реальных размеров окна, иначе толщина вертикальных и гори- зонтальных линий может существенно отличаться.

Логические размеры окна определяются функцией SetWindowExtEx(): BOOL SetWindowExtEx(HDC hdc, int x, int y, LPSIZE size);

Функция принимает контекст устройства hdc и логические размеры x и y, воз- вращает ненулевое значение при успешном завершении. Если последним па- раметром передан адрес структуры SIZE, то там будет сохранена информация о предыдущем размере окна, если она не нужна, указывается NULL.

##### Определение области вывода

Для режимов отображения MM\_ISOTROPIC и MM\_ANISOTROPIC необходимо указать и физический размер области экрана в пикселах. Этот размер может быть как меньше, так и больше фактического размера окна и определяется вызовом функции SetViewportExtEx():

BOOL SetViewportExtEx(HDC hdc, int x, int y, LPSIZE size);

Эта функция имеет такой же набор параметров, что и предыдущая, но x,y

сейчас определяет физические размеры области вывода.



Задание размера окна отрицательным числом приводит к изменению направ- ления оси координат.

##### Задание начала системы координат

Осталось определить новое начало системы координат. По умолчанию нача- ло координат, т. е. точка с координатами (0; 0), находится в левом верхнем углу окна (точнее клиентской области). Мы можем разместить начало коор- динат в любой точке области вывода при помощи функции SetViewportOrgEx():

BOOL SetViewportOrgEx(HDC hdc, int x, int y, LPPOINT point);

Функция принимает контекст устройства hdc, координаты нового начала ко- ординат (x,y) в экранных координатах и указатель на структуру point для хранения старого начала координат. Если последний параметр NULL, преды- дущее начало координат не сохраняется.

Для демонстрации использования последних рассмотренных функций пере- пишем в программе вывода графика синусоиды обработчик сообщения WM\_PAINT. Текст нового обработчика сообщения WM\_PAINT для программы по- строения графика функции с определением логических размеров окна и пе- реносом начала координат приведен в листинге 5.8.

**Листинг 5.8**

case WM\_PAINT : hdc = BeginPaint(hwnd,&ps);

GetClientRect(hwnd, &rt);

// Установка режима

**SetMapMode(hdc,MM\_ANISOTROPIC);**

// Установка логических размеров вывода, ось y направлена вверх

**SetWindowExtEx(hdc,628,-400,NULL);**

// Установка физических размеров на все окно

**SetViewportExtEx(hdc,rt.right,rt.bottom,NULL);**

// Установка начала координат в центр окна

**SetViewportOrgEx(hdc,rt.right/2,rt.bottom/2,NULL);**

// Сейчас работаем в логических координатах

SelectObject(hdc,hpen1); MoveToEx(hdc,-314,0,NULL); LineTo(hdc,314,0); MoveToEx(hdc,0,200,NULL); LineTo(hdc,0,-200); SelectObject(hdc,hpen2); x\_scr = -314; MoveToEx(hdc,x\_scr,0,NULL);

for (x = -PI; x < PI; x += 0.01, x\_scr++)

{

y = sin(x); y\_scr = y\*200;

LineTo(hdc, x\_scr, y\_scr);

}

EndPaint(hwnd,&ps); break;

Преобразование системы координат мы выделили жирным шрифтом. Выбран режим MM\_ANISOTROPIC и логические размеры [628; 400], а физические разме-

ры — фактический размер окна. Поскольку при указании логических разме- ров мы задали размер по вертикали отрицательным числом, то ось y новой системы координат сейчас направлена, как и принято, вверх. Начало же ко- ординат мы установили в центр окна (рис. 5.12).



|  |  |
| --- | --- |
|  |  |
|  |  |

**Рис. 5.12.** Преобразование системы координат

Если сравнить текст нового обработчика сообщения с предыдущим, то можно убедиться, что вычисления упростились и стали более наглядными.

#### Создание кисти

Кисть используется для заполнения фона всего окна при его создании или замкнутой области внутри окна. Сплошная кисть создается при помощи функции CreateSolidBrush():

HBRUSH CreateSolidBrush(COLORREF color);

и устанавливается функцией SelectObject(), например:

HBRUSH hbrush = CreateSolidBrush(RGB(255, 255, 0)); SelectObject(hdc,hbrush);

Штриховая кисть создается функцией CreateHatchBrush(): HBRUSH CreateHatchBrush(int nIndex, COLORREF color);

где первый параметр nIndex определяет тип штриховки и может принимать следующие значения:

* HS\_BDIAGONAL — слева направо и снизу вверх;
* HS\_CROSS — горизонтальная и вертикальная штриховка;
* HS\_DIAGCROSS — под углом в 45 градусов;
* HS\_FDIAGONAL — слева направо и сверху вниз;
* HS\_HORIZONTAL — горизонтальная штриховка;
* HS\_VERTICAL — вертикальная штриховка.

Второй параметр, как и раньше, обозначает цвет линии штриховки. Пример создания штриховой кисти:

HBRUSH hbrush = CreateHatchBrush(HS\_CROSS,RGB(0, 128, 0)); SelectObject(hdc,hbrush);

Кисть также должна быть удалена при завершении работы программы функ- цией DeleteObject().



Перед удалением кисти ее нужно освободить заданием новой кисти или опре- делением стандартной кисти.

Имеется набор системных кистей, которые могут устанавливаться функцией GetStockObject(), как мы это сделали, определив стандартную кисть при соз- дании окна в одном из следующих макросов:

* BLACK\_BRUSH — черная кисть;
* DKGRAY\_BRUSH — темно-серая кисть;
* GRAY\_BRUSH — серая кисть;
* LTGRAY\_BRUSH — светло-серая кисть;
* NULL\_BRUSH — нулевая кисть;
* WHITE\_BRUSH — белая кисть.

Системные кисти не нужно удалять при завершении программы.

После всего вышеизложенного нам будет несложно написать простую тесто- вую программу (листинг 5.9), которая показывает заполнение графических объектов сплошной и штриховыми кистями.

**Листинг 5.9**

HBRUSH hbrush,h\_brush[6];

char \*str = "сплошное заполнение";

char \*hstr[] = {"HS\_BDIAGONAL – слева направо и снизу вверх", "HS\_CROSS - горизонтальная и вертикальная штриховка", "HS\_DIAGCROSS - под углом в 45 градусов", "HS\_FDIAGONAL – слева направо и сверху вниз", "HS\_HORIZONTAL - горизонтальная штриховка", "HS\_VERTICAL - вертикальная штриховка"};

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

{

HDC hdc; PAINTSTRUCT ps;

int i;

int nIndex[] = {HS\_BDIAGONAL,HS\_CROSS,HS\_DIAGCROSS, HS\_FDIAGONAL,HS\_HORIZONTAL,HS\_VERTICAL};

switch(message)

{

case WM\_CREATE :hbrush = CreateSolidBrush(RGB(255, 255, 0)); for (i = 0; i < 6; i++) h\_brush[i]=CreateHatchBrush(nIndex[i],RGB(0, 128, 0)); break;

case WM\_PAINT : hdc = BeginPaint(hwnd,&ps); SelectObject(hdc,hbrush); Ellipse(hdc,1,1,40,40); TextOut(hdc,50,11,str,strlen(str)); for (i = 0; i < 6; i++)

{

SelectObject(hdc,h\_brush[i]); Rectangle(hdc,1,41+i\*40,40,80+i\*40); TextOut(hdc,50,51+i\*40,hstr[i],strlen(hstr[i]));

}

EndPaint(hwnd,&ps); break;

case WM\_SIZE :InvalidateRect(hwnd,NULL,1); break; case WM\_DESTROY :DeleteObject(hbrush);

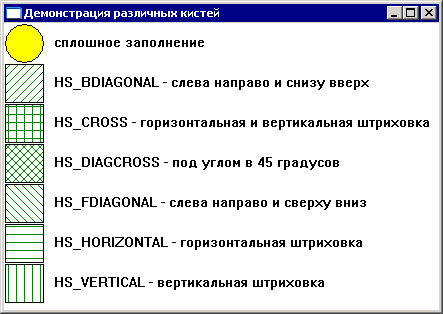
for (i = 0; i < 6; i++) DeleteObject(h\_brush[i]);

PostQuitMessage(0); break;// Завершение программы default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}



**Рис. 5.13.** Заполнение различными кистями

Здесь мы определили дескрипторы кистей и текстовые строки на глобальном уровне, причем определили их в виде массивов, что позволило организовать цикл для перебора всех 6 штриховых кистей.

В результате работы программы получим картинку, изображенную на рис. 5.13.

### Работа с текстом

До сих пор мы выводили текст, используя стандартный шрифт по умолча- нию. Сейчас посмотрим, как можно установить другой шрифт для вывода текста, а также использовать различные возможности его оформления.

#### Цвет текста и фона

При выводе текста функцией TextOut() имеется возможность установить цвет фона и текста при помощи пары функций: SetBkColor() и SetTextColor():

COLORREF SetBkColor(HDC hdc, COLORREF color); COLORREF SetTextColor(HDC hdc, COLORREF color);

Обе функции принимают дескриптор контекста устройства hdc и цвет color, а возвращают предыдущий цвет или CLR\_INVALID в случае ошибки.



Функция SetBkColor() определяет также фон между линиями для штриховой кисти.

Имеется еще одна функция SetBkMode(), которая управляет режимом отобра- жения фона при выводе текста:

int SetBkMode(HDC hdc, int mode);

Эта функция также получает дескриптор контекста устройства hdc и mode — режим отображения, принимающий одно из двух значений:

* OPAQUE — цвет фона при выводе текста будет определяться текущим, за- данным функцией SetBkColor(), установлено по умолчанию;
* TRANSPARENT — цвет фона при выводе текста не изменится. Например, так можно установить вывод синего текста на желтом фоне:

SetBkColor(hdc,RGB(255,255,0)); SetTextColor(hdc,RGB(0,0,128));

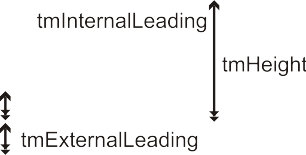
#### Получение метрики текста

При помощи функции GetTextMetrics() можно получить информацию о те- кущем шрифте, установленном для окна. Функция получает дескриптор кон-

текста устройства и заполняет структуру TEXTMETRIC информацией о текущем шрифте.

BOOL GetTextMetrics(HDC hdc, TEXTMETRIC\* tm);

Переменная имеет множество полей, содержащих характеристики шрифта, но в настоящее время нас интересуют лишь те, которые характеризуют размер шрифта (рис. 5.14).



**Рис. 5.14.** Основные характеристики шрифта

Построим тестовую программу для демонстрации текстового вывода шриф- том по умолчанию (листинг 5.10). Покажем различное фоновое заполнение, установку цвета текста и выведем основные характеристики шрифта.

**Листинг 5.10. Вывод метрики шрифта, заданного по умолчанию**

char str[] = "Текст для вывода в окне";

void Outtext(HDC hdc, int y, long tm, char \*height)

{

char rw[40],s[10];

ltoa(tm,s,10); strcpy(rw,height); strcat(rw,": "); strcat(rw,s);

TextOut(hdc,1,y,rw,strlen(rw));

}

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

{

HDC hdc; PAINTSTRUCT ps; TEXTMETRIC tm;

switch(message)

{

case WM\_PAINT : // Перерисовка экрана hdc = BeginPaint(hwnd,&ps);

SetBkColor(hdc,RGB(255,255,0));// Желтый фон SetTextColor(hdc,RGB(0,0,128));// Синий шрифт

TextOut(hdc,1,1,str,strlen(str)); SetBkMode(hdc,TRANSPARENT); // Прозрачный фон GetTextMetrics(hdc,&tm); Outtext(hdc,20,tm.tmHeight,"tmHeight"); Outtext(hdc,40,tm.tmInternalLeading,"tmInternalLeading"); Outtext(hdc,60,tm.tmExternalLeading,"tmExternalLeading"); Outtext(hdc,80,tm.tmAscent,"tmAscent"); Outtext(hdc,100,tm.tmDescent,"tmDescent"); EndPaint(hwnd,&ps);

break;

case WM\_SIZE :InvalidateRect(hwnd,NULL,1); break; case WM\_DESTROY : PostQuitMessage(0);

break; // Завершение программы

default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}

Здесь использована написанная нами небольшая функция Outtext() — чтобы не писать один и тот же код 5 раз, вместо этого обращаемся к этой функции и передаем ей необходимые параметры, в том числе и поле структуры TEXTMETRIC для вывода в окне его значения.

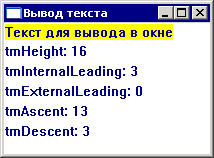
Обратите внимание, что первая строка на рис. 5.15 отображается с жел- тым фоном, поскольку по умолчанию работает параметр OPAQUE функции SetBkMode(). Однако, после того как мы установили параметр TRANSPARENT вызовом функции:

SetBkMode(hdc,TRANSPARENT);

то остальной вывод будет идти на белом фоне.

Отметим, в заключение, что высота шрифта определяется суммой высоты букв и межстрочным расстоянием:

Height = tmHeight + tmExternalLeading;



**Рис. 5.15.** Вывод характеристик шрифта

#### Определение длины строки

Иногда при выводе текста необходимо определить длину строки в логиче- ских единицах. Это бывает нужно или для определения позиции последую- щего вывода, или для организации переноса строки, когда она не входит в одну экранную строку. Задача не столь проста, как может показаться на пер- вый взгляд, поскольку если шрифт не является моноширинным, то символы имеют различную ширину. К счастью, имеется специальная функция, которая выполнит за нас эту работу:

BOOL GetTextExtentPoint32W(HDC hdc, LPCWSTR str, int len,

LPSIZE size);

где hdc — дескриптор контекста устройства, str — выводимая строка, len — длина этой строки и size — указатель на структуру SIZE:

struct SIZE

{ LONG cx; // Ширина LONG cy; // Высота

};

Так, если мы допишем в нашу предыдущую программу пару строк:

SIZE size; GetTextExtentPoint32(hdc,str,strlen(str),&size);

то из переменных size.cx и size.cy можно извлечь и вывести в окно значе- ния ширины и высоты строки str. Проделайте это самостоятельно.

#### Системные шрифты

Операционная система имеет набор встроенных шрифтов, которые может использовать прикладная программа, выбрав их при помощи функции GetStockObject(), а затем переключиться на нужный шрифт функцией SelectObject(). В табл. 5.5 приведены макросы, определяющие эти шрифты.

***Таблица 5.5.*** *Макроимена стандартных шрифтов*

|  |  |
| --- | --- |
| **Макрос** | **Шрифт** |
| ANSI\_FIXED\_FONT | Шрифт с фиксированным размером символов |
| ANSI\_VAR\_FONT | Шрифт с переменной шириной символов |
| DEVICE\_DEFAULT\_FONT | Шрифт по умолчанию |
| DEFAULT\_GUI\_FONT | Шрифт графического интерфейса по умолчанию |
| OEM\_FIXED\_FONT | OEM – шрифт (кириллицы нет) |

***Таблица 5.5*** *(окончание)*

|  |  |
| --- | --- |
| **Макрос** | **Шрифт** |
| SYSTEM\_FONT | Системный шрифт |
| SYSTEM\_FIXED\_FONT | Системный шрифт (устаревший) |

Для тестирования стандартных шрифтов допишем в начало предыдущей про- граммы несколько строк:

HFONT oldFont, newFont;

newFont = (HFONT)GetStockObject(ANSI\_FIXED\_FONT); oldFont = (HFONT)SelectObject(hdc,newFont);

Первой строкой мы определяем дескрипторы newFont для нового и oldFont для предыдущего шрифта. Обычно стараются сохранять дескриптор преды- дущего объекта, чтобы иметь возможность к нему вернуться, но, если такой потребности нет, можно использовать функцию SelectObject() без возвра- щаемого значения:

SelectObject(hdc, oldFont);

Если добавить эти три строки перед выводом текста, то текст будет выво- диться выбранным шрифтом. Меняя макроопределения различных шрифтов, можно посмотреть, чем отличается выводимый текст, а также его метрика.



Обе функции GetStockObject() и SelectObject() требуют явного преобразо- вания типа, в нашем случае к (HFONT).

Освобождать ресурсы, занятые встроенными шрифтами, не нужно.

#### Определение произвольных шрифтов

Кроме встроенных шрифтов, в программе может использоваться любой шрифт, зарегистрированный в системе. Для того чтобы программа могла осуществлять вывод выбранным шрифтом, его необходимо создать функцией CreateFont():

HFONT WINAPI CreateFont(int Height, int Width, int Escapement,

int Orientation, int Weight, DWORD Ital,

DWORD Underline, DWORD StrikeThru, DWORD Charset, DWORD Precision, DWORD ClipPrecision,

DWORD Quality, DWORD Pitch, LPCSTR FontName);

где необходимо задать значение параметров:

* Height — высота шрифта;
* Width — ширина шрифта;
* Escapement — угол наклона строки текста относительно горизонтальной оси в десятых долях градуса;
* Orientation — угол наклона каждого символа относительно горизонталь- ной оси в десятых долях градуса;
* Weight — насыщенность (жирность) текста, лежит в диапазоне [0; 1000]. (400 — нормальный текст, 700 — жирный.) FW\_NORMAL или 0 задает нор- мальную жирность;
* Ital — ненулевое значение создает наклонный шрифт;
* Underline — ненулевое значение создает подчеркнутый шрифт;
* StrikeThru — ненулевое значение создает перечеркнутый шрифт;
* Charset — определяет множество символов шрифта, обычно задается мак- росом, например DEFAULT\_CHARSET;
* Precision — задает точность отображения шрифта, определяется макро- сом, например OUT\_DEFAULT\_PRECIS;
* ClipPrecision — определяет, как будут отсекаться символы, не попадаю- щие в видимую область вывода, при помощи макроса, например CLIP\_DEFAULT\_PRECIS;
* Quality — качество шрифта: DEFAULT\_QUALITY, DRAFT\_QUALITY, PROOF\_QUALITY;
* Pitch — тип и семейство шрифтов. Значение формируется операцией ло- гического сложения для выбранного типа и семейства.

Тип: DEFAULT\_PITCH, FIXED\_PITCH, VARIABLE\_PITCH.

Семейство: FF\_DECORATIVE, FF\_DONTCARE, FF\_MODERN, FF\_ROMAN, FF\_SCRIPT, FF\_SWISS;

* FontName — указатель на строку, содержащую имя шрифта (до 32 сим- волов).



Функция CreateFont()не создает шрифт, а лишь настраивает существующий в системе в соответствии с заданными параметрами.

Так, если мы в предыдущей программе заменим строки выбора шрифта на следующие:

newFont = CreateFont(24,0,0,0,FW\_NORMAL,0,0,0,

DEFAULT\_CHARSET,OUT\_DEFAULT\_PRECIS,CLIP\_DEFAULT\_PRECIS, DEFAULT\_QUALITY, DEFAULT\_PITCH | FF\_DONTCARE,

"Times New Roman"); oldFont = (HFONT)SelectObject(hdc,newFont);

то в результате получим окно, изображенное на рис. 5.16, где мы добавили еще несколько строк кода для вывода длины и ширины выводимого текста:

SIZE size;

char s[30], ls[10];

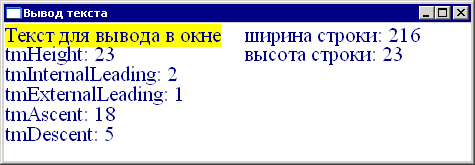
// Определение размеров строки текста в пикселах GetTextExtentPoint32(hdc,str,strlen(str),&size); strcpy(s,"ширина строки: "); ltoa(size.cx,ls,10);

strcat(s,l);

TextOut(hdc,240,1,s,strlen(s)); // Вывод ширины строки strcpy(s,"высота строки: ");

ltoa(size.cy,ls,10); strcat(s,l);

TextOut(hdc,240,20,s,strlen(s)); // Вывод высоты строки



**Рис. 5.16.** Демонстрация вывода шрифтом "Times New Roman"

Однако перед завершением обработки сообщения необходимо освободить ресурсы, занятые созданным шрифтом. Но мы не можем удалить текущий шрифт, поэтому обычно вначале восстанавливают старый шрифт, и лишь за- тем удаляют вновь созданный:

SelectObject(hdc,oldFont); // Восстановить шрифт по умолчанию DeleteObject(newFont); // Освободить ресурсы



Здесь мы создаем новый шрифт при каждом вхождении в обработчик сообще- ния WM\_PAINT, поэтому и вынуждены при завершении обработки сообщения удалить созданный шрифт, иначе мы "замусорим" память "бесхозными" шриф- тами. Однако часто поступают иначе, и создают шрифты при создании окна (сообщение WM\_CREATE), а уничтожают при его закрытии (сообщение WM\_DESTROY).

Напишем еще одну небольшую программу (листинг 5.11) для демонстрации использования произвольных шрифтов. Используем возможность изменения ориентации выводимой строки, параметр Orientation. Изменим также насы- щенность шрифта Weight и наклон Ital.

**Листинг 5.11. Вывод текста по радиальным линиям**

char str[] = "Наклонный текст";

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

{

HDC hdc;

HFONT oldFont, newFont; PAINTSTRUCT ps;

RECT rt;

int i, a, b; switch(message)

{

case WM\_PAINT :

hdc = BeginPaint(hwnd,&ps); GetClientRect(hwnd, &rt); a = rt.right/2;

b = rt.bottom/2;

for (i = 0; i < 3600; i += 200)

{

newFont = CreateFont(20,0,i,0,700,1,0,0,

DEFAULT\_CHARSET, OUT\_DEFAULT\_PRECIS, CLIP\_DEFAULT\_PRECIS, DEFAULT\_QUALITY, DEFAULT\_PITCH | FF\_DONTCARE,"Arial");

oldFont = (HFONT)SelectObject(hdc,newFont); TextOut(hdc,a,b,str,strlen(str)); SelectObject(hdc,oldFont); DeleteObject(newFont);

}

EndPaint(hwnd,&ps); break;

case WM\_SIZE :InvalidateRect(hwnd,NULL,1); break; case WM\_DESTROY : PostQuitMessage(0); break;

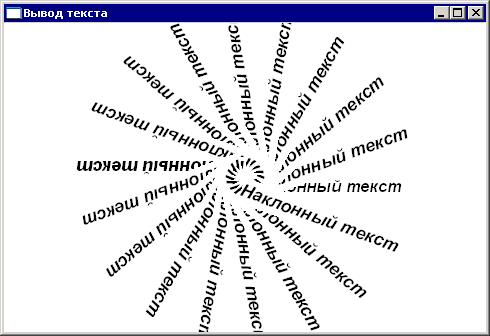
default : return DefWindowProc(hwnd, message, wParam, lParam);

}

return 0;

}

Результат работы программы приведен на рис. 5.17.



**Рис. 5.17.** Вывод текста по наклонным линиям из центра окна

## Диалог с пользователем

Вся идеология построения Windows-программы ориентирована на взаимо- действие с пользователем. Это может быть выбор пункта меню, специальные окна для ввода или выбора данных и т. п. Наиболее простым элементом ин- терфейса является *окно сообщений*.

### Окно сообщений

Часто возникает ситуация, когда программа должна получить согласие на выполнение действий. Для этих целей используется API-функция MessageBox(), которая отображает сообщение и ждет реакции пользователя. Никаких до- полнительных действий предпринимать не нужно, функция сама создаст и отобразит окно, а после ответа пользователя его уничтожит.

int WINAPI MessageBox(HWND hwnd, LPCSTR Text, LPCSTR Caption, UINT Type);

где параметры:

* hwnd — дескриптор родительского окна;
* Text — указатель строки сообщения;
* Caption — указатель строки заголовка окна сообщения;
* Type — определяет свойства окна сообщения и строится как логическая сумма этих свойств.

Вот некоторые типичные его значения:

 MB\_OK Отображается кнопка OK.

 MB\_OKCANCEL Отображаются кнопки OK и CANCEL.

 MB\_ABORTRETRYIGNORE Отображаются кнопки ABORT, RETRY и IGNORE.  MB\_YESNOCANCEL Отображаются кнопки YES, NO и CANCEL.

 MB\_YESNO Отображаются кнопки YES и NO.

 MB\_RETRYCANCEL Отображаются кнопки RETRY и CANCEL.  MB\_ICONHAND Отображается иконка " ".

 MB\_ICONQUESTION Отображается иконка "?".

 MB\_ICONEXCLAMATION Отображается иконка "!".  MB\_ICONASTERISK Отображается иконка "\*".

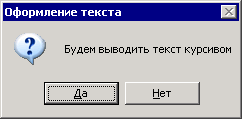
Так, если в предыдущем примере мы хотели бы перед выводом текста спро- сить пользователя, хочет ли он выводить текст курсивом, то мы могли бы это реализовать перед определением шрифта следующим образом:

int k;

i = MessageBox(hwnd, "Будем выводить текст курсивом", "Оформление текста", MB\_YESNO | MB\_ICONQUESTION);

k = (i == IDYES)? 1 : 0;

а шестым параметром Ital функции CreateFont() вместо 1 укажем перемен- ную k. В результате мы увидим диалоговое окно, изображенное на рис. 5.18.



**Рис. 5.18.** Диалоговое окно MessageBox()

При нажатии кнопки "Да" функция возвратит значение IDYES, и в этом случае мы присваиваем переменной k значение 1, иначе — 0. Так что, в зависимости от ответа пользователя, текст будет выводиться либо курсивом, либо нор- мальным шрифтом.

### Меню

Ни одна Windows-программа не обходится без *меню*. При помощи выполне- ния команд меню происходит открытие и закрытие файлов, выбор и настрой-

ка устройства печати, собственно настройки программы и т. п. Причем во всех Windows-программах при создании меню необходимо придерживаться некоторых устоявшихся традиций. Так меню верхнего уровня всегда отобра- жается вверху под заголовком окна, подменю отображаются как выпадающие меню.

Для создания меню необходимо использовать специальный файл — *файл ре- сурсов*, который для меню является просто текстовым файлом определенной структуры.

Хотя среда программирования предоставляет для этой цели определенный механизм, мы пока не будем его использовать. Проще написать этот файл "вручную", используя, например, Блокнот. Имя файла произвольно, но рас- ширение должно быть rc, месторасположение файла — в папке проекта.

Определение меню имеет следующий вид:

MenuName MENU параметр

{

элементы меню

}

Значения параметров меню приведены в табл. 5.6.

***Таблица 5.6.*** *Значения параметров меню*

|  |  |
| --- | --- |
| **Параметр** | **Назначение** |
| DISCARDABLE | Неиспользуемое меню может быть удалено из памяти |
| FIXED | Меню фиксировано в памяти |
| LOADONCALL | Меню загружается перед использованием |
| MOVEABLE | Меню может перемещаться в памяти |
| PRELOAD | Меню загружается при старте программы |

Имеется два типа элементов меню: POPUP — выпадающее меню, в котором могут содержаться любые элементы меню; MENUITEM — обычный элемент меню:

MENUITEM "Имя", MenuID[, параметры] POPUP "Имя", MenuID[, параметры]

Имя — это текст, отображаемый в пункте меню, MenuID — уникальный цело- численный идентификатор, который обычно определяется в заголовочном файле. [параметры] являются необязательной частью конструкции и позволя-

ют делать пункты меню неактивными, определяют способ размещения пунк- тов и т. п. Здесь мы не будем углубляться в подробности.

Рассмотрим создание меню на примере рисования прямоугольника или эл- липса, где мы будем в меню задавать тип фигуры, толщину линии и цвет за- полнения.

Для создания меню можно воспользоваться любым текстовым редактором, но сохранить в текстовом файле с именем menu.rc (листинг 5.12) в папке про- екта, а затем добавить его к проекту командой контекстного меню правой кнопки мыши **Add Files to Folder**. При этом курсор мыши должен показы- вать на папку в рабочей области (Workspace).

|  |
| --- |
| **Листинг 5.12. Файл menu.rc** |

#include "menu.h" MAINMENU MENU DISCARDABLE

{

POPUP "Фигура"

{

MENUITEM "Прямоугольник", RECTANGLE MENUITEM "Эллипс", ELLIPSE

}

POPUP "Линия"

{

MENUITEM "Тонкая", THIN

MENUITEM "Удвоенной толщины", THICK

}

POPUP "Цвет заливки"

{

MENUITEM "Красный", RED MENUITEM "Зеленый", GREEN MENUITEM "Синий", BLUE

}

}

Это меню с именем MAINMENU содержит три пункта верхнего уровня и выпа- дающие меню из двух и трех пунктов.

Поскольку мы определили идентификаторы пунктов меню, то необходимо добавить файл включений menu.h (листинг 5.13), где эти идентификаторы бу- дут определены. Этот файл можно добавить к проекту, выполнив команду **File | New | C/C++ Header File**.

**Листинг 5.13. Файл menu.h**

#define RECTANGLE 1

#define ELLIPSE 2

#define THIN 3

#define THICK 4

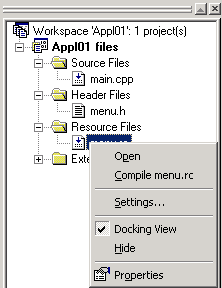
#define RED 5

#define GREEN 6

#define BLUE 7

Этот файл необходимо включить и в программу, использующую меню.

После создания файла ресурса и заголовочного файла для него необходимо скомпилировать меню (рис. 5.19), поскольку соединение его с Windows- программой осуществляется на этапе компоновки. Это можно сделать, выде- лив файл ресурса при помощи контекстного меню, вызываемого нажатием правой кнопки мыши.



**Рис. 5.19.** Контекстное меню для файлов ресурса

После компиляции файла ресурсов в папке Debug будет создан компилиро- ванный файл menu.res, который и будет использоваться компоновщиком для построения рабочей программы.

#### Обработка команд меню

При выборе пункта меню операционная система посылает окну сообщение

WM\_COMMAND. Идентификатор же выбранного пункта меню содержится в млад-

шем слове параметра wParam, и для его извлечения можно использовать мак- рос LOWORD(wParam). Обычно для обработки пунктов меню используют пере- ключатель, например:

case WM\_COMMAND :

switch(LOWORD(wParam))

{

case ALPHA : ...

case BETA : ...

case GAMMA : ...

}

break;

Реализуем теперь обработку пунктов меню, показанного в листинге 5.14.

|  |
| --- |
| **Листинг 5.14. Программа рисования прямоугольника или эллипса с использованием меню** |

#include "menu.h"

LRESULT CALLBACK WindowsFunc(HWND hwnd, UINT message,

WPARAM wParam, LPARAM lParam)

{

HDC hdc; PAINTSTRUCT ps; RECT rt;

int a,b;

static int figure, width; static COLORREF color; HPEN oldpen, hpen; HBRUSH oldbrush, hbrush; switch(message)

{

case WM\_CREATE :figure = 0; width = 1;

color = RGB(255,255,255);

break; case WM\_COMMAND :

switch(LOWORD(wParam))

{

case RECTANGLE: figure = 0; break; case ELLIPSE : figure = 1; break; case THIN : width = 1; break; case THICK : width = 2; break;

case RED : color = RGB(255,0,0); break; case GREEN : color = RGB(0,255,0); break; case BLUE : color = RGB(0,0,255);

}

InvalidateRect(hwnd,NULL,1); break;

case WM\_PAINT :

hdc = BeginPaint(hwnd,&ps); GetClientRect(hwnd, &rt);

a = rt.right; // Ширина окна b = rt.bottom; // Высота окна

hpen = CreatePen(PS\_SOLID,width,0); oldpen = (HPEN)SelectObject(hdc,hpen); hbrush = CreateSolidBrush(color);

oldbrush = (HBRUSH)SelectObject(hdc,hbrush);

if (figure == 0) Rectangle(hdc,a/4,b/4,3\*a/4,3\*b/4); else Ellipse(hdc,a/4,b/4,3\*a/4,3\*b/4); SelectObject(hdc,oldpen); SelectObject(hdc,oldbrush);

DeleteObject(hpen); DeleteObject(hbrush); EndPaint(hwnd,&ps); break;

case WM\_SIZE :InvalidateRect(hwnd,NULL,1); break; case WM\_DESTROY : PostQuitMessage(0); break;

default : return DefWindowProc(hwnd, message, wParam, lParam);

}

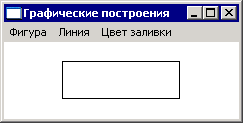
return 0;

}

Хотя текст обработчика сообщения готов, мы еще сделали не все для того, чтобы можно было компилировать программу. Предварительно необходимо подключить меню к оконному классу, т. е. присвоить полю lpszMenuName пе- ременной класса окна имя меню:

wc.lpszMenuName = "mainmenu";

Теперь можно компилировать программу и, в случае отсутствия ошибок, вы- полнить, получив результат такой же, как показан на рис. 5.20.



**Рис. 5.20.** Программа с главным меню

Прокомментируем приведенную в листинге 5.14 программу.

Мы должны описать переменные, которые характеризуют выводимый объект:

* figure — тип фигуры;
* width — толщина линии;
* color — цвет заполнения;
* oldpen, hpen — дескрипторы пера;
* oldbrush, hbrush — дескрипторы кисти.

Поскольку первые три переменных должны сохранять свое значение при вы- ходе из оконной функции, мы их описываем как статические:

static int figure, width; static COLORREF color;

Дескрипторы мы определяем только в одном обработчике, поэтому они могут быть автоматическими переменными. По два дескриптора мы определим, чтобы иметь возможность удалить созданные перо и кисть, не замусорив память.

Так мы создадим перо и кисть:

hpen = CreatePen(PS\_SOLID,width,0); // Определяем перо oldpen = (HPEN)SelectObject(hdc,hpen); // Сохраняем старое перо hbrush = CreateSolidBrush(color); // Определяем кисть

oldbrush = (HBRUSH)SelectObject(hdc,hbrush); // Сохраняем старую кисть

Поскольку нельзя удалять текущее перо и кисть, нам приходится перед уда- лением вернуться к значению пера и кисти по умолчанию:

SelectObject(hdc,oldpen); // Выбрать старое перо SelectObject(hdc,oldbrush); // Выбрать старую кисть DeleteObject(hpen); // Удалить созданное перо DeleteObject(hbrush); // Удалить созданную кисть

Выбор фигуры для рисования осуществляем оператором if () ... else ...

Быть может, это не самое лучшее решение, но самое простое:

if (figure == 0) Rectangle(hdc,a/4,b/4,3\*a/4,3\*b/4); else Ellipse(hdc,a/4,b/4,3\*a/4,3\*b/4);

Фигура строится в ¼ окна.

Для того чтобы фигура сразу была построена при создании окна, в сообще- нии WM\_CREATE мы устанавливаем начальное значение параметров:

case WM\_CREATE :figure = 0; width = 1;

color = RGB(255,255,255);

break;

При обработке сообщения WM\_COMMAND, т. е. при выборе пункта меню, мы про- сто изменяем значения параметров figure, width или color и вызываем функ- цию InvalidateRect(), инициирующую перерисовку окна.

## Заключение

На этом мы завершим рассмотрение оконного интерфейса. Мы не ставили перед собой задачу познакомиться со всеми аспектами создания Windows- приложений, поэтому просто провели небольшой обзор для того, чтобы у чи- тателя сложилось некоторое представление о программировании оконного интерфейса. Дело в том, что в настоящее время практически уже никто не пишет программ, основанных на вызовах API-функций. Давно уже создана библиотека MFC, которая представляет библиотеку классов и, по сути, явля- ется обложкой для тех же API-функций, но помимо этого предоставляет мас- су дополнительных возможностей, что существенно облегчает процесс соз- дания полноценного приложения. К тому же и среда программирования MS Visual C++ 6.0 ориентирована на использование MFC и программирова- ние без этих библиотек становится все менее привлекательным занятием.

Мы же преследовали несколько иную цель — показать "изнутри", как осуще- ствляется обработка сообщений. При программировании же с MFC мы даже не имеем возможности увидеть головную функцию WinMain(), поскольку она скрыта в среде разработки. В этом есть определенный смысл, действительно, если эта функция практически не меняется, т. е. смысл использовать шаблон этой функции на все случаи жизни, изменяя лишь некоторые ее параметры по мере необходимости. Однако начинать изучение оконного интерфейса с биб- лиотеки MFC довольно проблематично — мы получим массу рецептов, но трудно добиться понимания, как же все это происходит.

## Вопросы к главе

1. Типы данных в Windows-приложениях.
2. Что такое дескриптор?
3. Что такое "функция с обратным вызовом"?
4. Какие действия выполняет головная функция WinMain()?
5. Назначение оконной функции.
6. Как происходит обработка сообщения о нажатии клавиши на клавиатуре?
7. В чем специфика сообщения WM\_PAINT?
8. Как извлечь координаты мыши при обработке сообщения о нажатии кнопки мыши?
9. Как заставить систему перерисовать окно?
10. Как создать и уничтожить таймер?
11. Функции рисования: параметры и возвращаемое значение.
12. Как определить размеры окна (клиентской области)?
13. Как обработать сообщение "Изменение размеров окна"?
14. Как создать и установить перо и кисть?
15. Зачем нужно уничтожать созданные перья и кисти при завершении рабо- ты приложения и как это сделать?
16. Что такое "Чистый цвет"?
17. Установка режима отображения.
18. Как определить физические и логические размеры окна и для каких ре- жимов это необходимо?
19. Как задать новую систему координат?
20. Установка фона и цвета при выводе текста.
21. Функция вывода текста.
22. Метрика текста.
23. Установка системного и произвольного шрифта.
24. Вывод окна сообщения.
25. Подключение пользовательского меню.

## Задание для самостоятельной работы

1. Написать программу, строящую прямоугольник в центре окна. При нажа- тии на левую кнопку мыши его размеры уменьшаются, а при нажатии на правую кнопку мыши его размеры, соответственно, увеличиваются на единицу.
2. То же самое, только размеры изменяются автоматически через 1 секунду. Нажатие на левую кнопку мыши меняет направление изменения размеров. Правая кнопка завершает работу.
3. Написать программу движения шарика в окне с отражением от стенок по законам геометрической оптики. Начало движения происходит из точки, в которой нажимается левая кнопка мыши. Скорость постоянна, началь- ный угол определяется случайным образом.
4. Написать программу, которая разрисует окно, как шахматную доску, и при нажатии клавиши мыши выведет окно сообщений с именем клетки, где находится курсор в шахматной нотации.
5. Создать тестовую программу вывода строки текста, меняя размер шрифта от минимально читаемого размера (определите опытным путем) до 1 дюйма.
6. Решите задачу о вычислении интеграла Френеля из *главы 1*, выведите в окне результат и дайте графическую иллюстрацию.

# Приложение

|  |
| --- |
| **Таблица приоритетов операций** |
| :: Доступ к области видимости  () Скобки  [] Выделение элемента массива  . Выделение элемента класса, структуры или объединения  -> То же по указателю |
| ! Логическое отрицание  ~ Побитовое отрицание  - Унарный минус  ++ Увеличение на 1  -- Уменьшение на 1  & Определение адреса  \* Обращение по адресу  new Выделение памяти  delete Освобождение памяти  (тип) Преобразование типа  sizeof Определение размера объекта в байтах |
| \* Умножение  / Деление  % Остаток от деления |
| + Сложение  - Вычитание |

*282 Приложение*

|  |
| --- |
| **Таблица приоритетов операций** *(окончание)* |
| << Сдвиг влево  >> Сдвиг вправо |
| < Меньше  <= Меньше или равно  > Больше  >= Больше или равно |
| == Равно  != Не равно |
| & Битовая операция И  ^ Битовая операция "исключающее ИЛИ"  | Битовая операция ИЛИ |
| && Логическая операция И  || Логическая операция ИЛИ |
| ?: Условная арифметическая операция |
| = Присваивание |
| throw Исключение |
| , Операция запятая |

# Рекомендуемая литература

1. Джосьютис Н. С++. Стандартная библиотека. Для профессионалов. — СПб.: Питер, 2004. — 730 с.: ил.
2. Лафоре Р. Объектно-ориентированное программирование в С++. Классика Computer Science. 4-е изд.— СПб.: Питер, 2003. — 928 с.: ил.
3. Павловская Т. А. С/С++. Программирование на языке высокого уровня. — СПб.: Питер, 2002. — 464 с.: ил.
4. Павловская Т. А., Шупак Ю. А. С/С++. Структурное программирование: Практикум. — СПб.: Питер, 2003. — 240 с.: ил.
5. Шилдт Г. Программирование на С и С++ для Windows 95. — К.: Торгово- издательское бюро BHV, 1996. — 400 с.: ил.

## Дополнительная литература

1. Круглински Д., Уингоу С., Шеферд Дж. Программирование на Microsoft Visual C++ 6.0 для профессионалов/Пер. с англ. — СПб.: Питер; М.: изд. "Русская редакция", 2003. — 864 с.: ил.
2. Эккель Б. Философия С++. Введение в стандартный С++. 2-е изд. — СПб.: Питер, 2004. — 572 с.: ил.
3. Эккель Б., Эллисон Ч. Философия С++. Практическое программирова- ние. — СПб.: Питер, 2004. — 608 с.: ил.

# Предметный указатель

### #

#define 10

#include 10

### A

adjustfield 119

advance() 207

ANSI 85

API 225

append() 215

ASCII 85

atof() 48

ATOM 226

auto 24

### B

back\_inserter() 206

bad() 128

badbit 127

base() 204

basefield 119

BeginPaint() 239

bind2nd() 149

bool 16

BOOL 226

break 34

BYTE 226

### C

c\_str() 212

CALLBACK 225

case 34

catch 107

char 16

CharToOem() 85

cin 33, 115

class 70

clear() 128

close() 125

COLORREF 226, 249

compare() 214

const 24

continue 34

copy\_backward() 157

cout 12, 115

CreateFont() 266

CreateHatchBrush() 259

CreatePen() 251

CreateSolidBrush() 259

CreateWindow() 233

### D

data() 212

default 34

DefWindowProc() 235

delete 43

DeleteObject() 253, 260

deque 183

DispatchMessage() 234

distance() 207

double 16, 20

DWORD 226

### E

Ellipse() 248

else 31

EndPaint() 239

enum 18

EOF 120

eof() 127

eofbit 127

equal\_range() 193

exception 107

### F

fail() 128

failbit 127

filebuf 132

fill() 120

find() 193

flags() 119

float 16, 19

FLOAT 226

floatfield 119

flush() 122

friend 85

front\_inserter() 206

fstream 125

### G

gcount() 120

get() 120

GetClientRect() 246

GetDC() 236

getline() 120

GetMessage() 234, 235

GetStockObject() 233, 253,

260

GetTextExtentPoint32W() 265

GetTextMetrics() 262

good() 128

goodbit 127

goto 30

### H

HANDLE 226

hardfail 127

HBRUSH 259

HDC 236

HFONT 266

HIWORD 241

HWND 231

### I

if 31

ifstream 125

ignore() 90, 121

in\_avail() 132

inline 51, 75

inserter() 206

int 16

INT 226

InvalidateRect() 241

ios 116

is\_open() 125

istream 86, 120

istream\_iterator 159

istreambuf\_iterator 133

istrstream 122

iter\_swap() 207

itoa() 243

### K

KillTimer() 244

### L

LineTo() 245

LoadCursor() 232

LoadIcon() 232

long 16

LONG 226

long double 16

lower\_bound() 193

LOWORD 241

LPARAM 226

LRESULT 226

### M

main() 11, 65

MENUITEM 272

MessageBox() 270

min 50

MM\_TEXT 256

MoveToEx() 245

MSG 231

### N

namespace 87

new 42

### O

OemToChar() 86

ofstream 125

open() 125

operator 86

ostream 86, 119

ostream\_iterator 158

ostreambuf\_iterator 132

ostrstream 122

### P

PAINTSTRUCT 240

pair 156

peek() 121

pop\_front() 185

POPUP 272

PostQuitMessage() 235

precision() 120

private 70

protected 90

public 70

push\_front() 185

put() 122

putback() 121

### Q

queue 210

### R

rand() 163

rdbuf() 131

rdstate() 127

read() 121

readsome() 121

RECT 246

Rectangle() 248

register 24

ReleaseDC() 237

return 49

reverse\_iterator 204

RGB() 249

### S

sbumpc() 132

seekg() 129

seekp() 129

SelectObject() 253

SetBkColor() 262

SetBkMode() 262

setf() 116, 119

setfill() 78

SetMapMode() 256

SetPixel() 249

SetTextColor() 262

SetTimer() 242

SetViewportExtEx() 257

SetViewportOrgEx() 257

setw() 78

SetWindowExtEx() 257

sgetc() 132

sgetn() 132

short 16

ShowWindow() 234

SIZE 265

sizeof 37

snextc() 132

sputbackc() 132

sputc() 132

stack 208

static 24, 79

std 88, 115

STL 137

strcpy() 53

strfill() 53

string 211

strlen() 52, 79, 238

strstream 122

strstreambuf 122

strtok() 47

struct 61

substr() 219

sungetc() 132

swap() 50, 55

switch 34

### T

tellg() 129

tellp() 129

template 101

TEXTMETRIC 263

TextOut() 236, 242

this 81

throw 108

TranslateMessage() 234

try 107

typedef 41

typename 101

### U

UINT 226

union 64

unsetf() 116

unsigned 16

upper\_bound() 193

using 88

### V

vector 138

virtual 80, 81, 96, 97

### W

width() 120

WinMain() 230

WM\_CHAR 236

WM\_COMMAND 274

WM\_CREATE 242, 277

WM\_DESTROY 235

WM\_LBUTTONDOWN 240

WM\_PAINT 242

WM\_QUIT 235

WM\_RBUTTONDOWN 240

WM\_SIZE 247

WNDCLASS 231

WORD 226

write() 122

### А

Алгоритм 137, 145

accumulate() 145, 179

adjacent\_difference() 182

adjacent\_find() 154

binary\_search() 173

copy() 145, 157

count() 145

cout() 155

cout\_if() 155

equal() 145, 155

equal\_range() 175

fill() 145, 162

fill\_n() 162

find() 145, 146

find\_end() 153

find\_first\_of() 153

find\_if() 146

for\_each() 145, 154

generate() 146, 163

generate\_n() 163

includes() 174

inner\_product() 180

inplace\_merge() 179

iter\_swap() 145 lexicographical\_compare() 156

lover\_bound() 174

max\_element() 146, 151

merge() 145, 176

min\_element() 146, 151

mismatch() 155

next\_permutation() 168

nth\_element() 173

partial\_sort() 172

partial\_sort\_copy() 172

partial\_sum() 181

partition() 146, 170

prev\_permutation() 168

random\_shuffle() 169

remove() 146, 164

remove\_copy() 165

remove\_copy\_if() 165

remove\_if() 164

replace() 146, 163

replace\_copy() 164

replace\_copy\_if() 164

replace\_if() 163

reverse() 146, 167

reverse\_copy() 167

rotate() 146, 167

rotate\_copy() 167

search() 145, 152

search\_n() 152

set\_difference() 178

set\_intersection() 177 set\_symmetric\_difference() 178

set\_union() 177

sort() 145, 171

stable\_partition() 170

stable\_sort() 171

swap() 145

swap\_ranges() 162

transform() 146, 160

unique() 146, 166

unique\_copy() 167

unique\_copy\_if() 167

upper\_bound() 175

численный 179

### Б

Байт 15

Бит 15

Битовое исключающее сложение 27

Битовое отрицание 27

Битовое умножение 27 Блок обработки ошибок 110

### В

Вектор 138

Вычитание 25

### Д

Двоичная система счисления 15

Дек 183

Деление 25

Дескриптор 236

 контекста устройства 236

Деструктор 80

 порожденного класса 98

### З

Защищенные члены класса 90

### И

Индекс 36

Инкапсуляция 70

Исключение 107

 Win32 API 108

Исключительная ситуация 107

Итератор 132, 137, 200

back\_inserter 160

ввода 200

вставки 205

вывода 200

двунаправленный 202

обратный 203

потоковый 157



произвольного доступа 203

прямой 201

### К

Класс 70

виртуальный 98

окна 226

памяти 24

родительский 100

синтаксис описания 70

шаблонный 102

Комментарий 24 Константа:

с плавающей точкой 21 символьная 21 символьная управляющая 21

строковая 22

текстовая 22

целая 20

Конструктор 73

 копирования 75

Контейнер 137

ассоциативный 137

последовательный 137

Контекст устройства 236

### Л

Лексема 47 Логический сдвиг:

влево 28

вправо 28

### М

Манипулятор 117

endl 117

ends 117

flush 118

setfill 118

setiosflags 118

setprecision 118

setw 117

ws 117

Мантисса 19

Массив 35

ассоциативный 198

многомерный 41

Меню 271

Метка 30

Метод 70, 118

виртуальный 96

чисто виртуальный 98

Множество 191

Мультимножество 191

Мультиотображение 195

### Н

Наследование 89, 91

 множественное 89

### О

Область видимости 78

Объединение 63

Окно сообщений 270 Оператор:

<< 122

>> 120

дружественный 85

логический 29

перехода 30

простой 30

преобразования типа 84

пустой 52

составной 30

условный 31 условный арифметический 31

 цикла 31

Операции над векторами: assign() 141

at() 142

back() 142

begin() 142

capacity() 139

clear() 143

empty() 139

end() 142

erase() 143

front() 142

insert() 143

max\_size() 139

pop\_back() 143

push\_back() 143

rbegin() 142

rend() 142

reserve() 139

resize() 143

size() 139

swap() 141

Операция 25

–– 27



++ 27

-> 62

битовая 27

взятия адреса 38

запятая 33

вычисление разности двух указателей 39

вычитание из указателя 39 комбинированная 28

отношения 29

постфиксная 27

префиксная 27

разадресации 38 сложение указателя с числом 39

 сравнение указателей 39 Остаток от деления 25 Отображение 195

Охраняемый раздел 108

Очередь 209

 приоритетная 210

### П

Перегрузка: методов 75

оператора 76

Переключатель 34 Переменные среды

окружения 66

Перечисляемый тип 18

Перо 251

Позднее связывание 97

Порядок 19

Предикат 147 Преобразование типа:

автоматическое 26

явное 26

Препроцессор 10

Приведение типов 26

Присваивание 25

Пространство имен 87

 стандартное 88

Прототип 10, 50

Псевдонимы имен 41

### С

Селектор выбора 62

Сложение 25

Сортировка 50

Список 187

Ссылочный тип данных 54 Стандартная библиотека

шаблонов 137

Стек 102, 208

Структура 61

### Т

Таймер 242

### У

Указатель 38

константный 39

на указатель 44

на функцию 52

Умножение 25

### Ф

Файл ресурсов 272

Флаг 116

app 126

ate 126

binary 126

boolalpha 116

dec 116

fixed 116

hex 116

in 125

internal 116

left 116

oct 116

out 125

right 116

scientific 116

showbase 116

showpoint 116

showpos 116

skipws 116

stdio 116

trunc 126

unitbuf 116

uppercase 116

Функциональный адаптер 149

Функциональный объект 147

divides<>() 148

equal\_to<>() 148

greater\_equal<>() 148

greater<>() 148

less\_equal<>() 148

less<>() 148

logical\_and<>() 148

logical\_not<>() 148

logical\_or<>() 148

minus<>() 148

modulus<>() 148

multiplies<>() 148

negate<>() 148

not\_equal\_to<>() 148

plus<>() 148

Функция 49

виртуальная 98

встраиваемая 51

### Ц

Цвет чистый 252 Цикл:



for 32

обработки сообщений 228 с постусловием do-while 32 с предусловием while 32

### Ш

Шаблон 100

класса 102

функции 100 Шестнадцатеричное

представление 15

Шрифт системный 265