

# Содержание

Благодарности	20
Об авторе	22
От издательства	22
Введение	23
Принятый подход	23
Примеры кода, используемые в книге	24
Организация книги	24
Примечание для преподавателей	28
Соглашения, используемые в этой книге	29
Системы, на которых разрабатывались примеры для книги	30
<b>Глава 1. Начало работы с C++</b>	<b>31</b>
Изучение языка C++: с чем придется иметь дело	32
Истоки языка C++: немного истории	33
Язык программирования C	33
Философия программирования на языке C	34
Переход к C++: объектно-ориентированное программирование	35
C++ и обобщенное программирование	37
Происхождение языка программирования C++	37
Переносимость и стандарты	38
Развитие языка	41
Эта книга и стандарты C++	41
Порядок создания программы	41
Создание файла исходного кода	42
Компиляция и компоновка	43
Компиляция и связывание в Unix	44
Компиляция и связывание в Linux	45
Компиляторы командной строки для режима командной строки Windows	45
Компиляторы для Windows	46
C++ в Macintosh	48
Резюме	48
<b>Глава 2. Приступаем к изучению C++</b>	<b>49</b>
Первые шаги в C++	50
Возможности функции <code>main()</code>	51
Заголовок функции как интерфейс	52
Почему функции <code>main()</code> нельзя назначать другое имя	54
Комментарии в языке C++	54
Препроцессор C++ и файл <code>iostream</code>	55
Имена заголовочных файлов	56
Пространства имен	57
Вывод в C++ с помощью <code>cout</code>	58
Манипулятор <code>endl</code>	60
Символ новой строки	60
Форматирование исходного кода C++	61
Лексемы и пробельные символы в исходном коде	61
Стиль написания исходного кода C++	62

Операторы в языке C++	63
Операторы объявления и переменные	63
Операторы присваивания	65
Новый трюк с объектом cout	66
Другие операторы C++	67
Использование cin	67
Конкатенация с помощью cout	68
cin и cout: признак класса	68
Функции	70
Использование функции, имеющей возвращаемое значение	70
Разновидности функций	74
Функции, определяемые пользователем	75
Использование определяемых пользователем функций, имеющих возвращаемое значение	78
Местоположение директивы using в программах с множеством функций	80
Резюме	81
Вопросы для самоконтроля	82
Упражнения по программированию	83
<b>Глава 3. Работа с данными</b>	<b>85</b>
Простые переменные	86
Имена, назначаемые переменным	87
Целочисленные типы	88
Целочисленные типы short, int, long и long long	89
Типы без знаков	94
Выбор целочисленного типа	96
Целочисленные литералы	97
Определение компилятором C++ типа константы	99
Тип char: символы и короткие целые числа	100
Тип bool	109
Квалификатор const	109
Числа с плавающей точкой	111
Запись чисел с плавающей точкой	111
Типы чисел с плавающей точкой	112
Константы с плавающей точкой	114
Преимущества и недостатки чисел с плавающей точкой	115
Арифметические операции в C++	116
Порядок выполнения операций: приоритеты операций и ассоциативность	117
Операция нахождения остатка от деления	119
Преобразования типов	120
Объявления auto в C++11	126
Резюме	127
Вопросы для самоконтроля	128
Упражнения по программированию	129
<b>Глава 4. Составные типы</b>	<b>131</b>
Введение в массивы	132
Замечания по программе	134
Правила инициализации массивов	135

## 8 Содержание

Инициализация массивов в C++11	136
Строки	136
Конкатенация строковых литералов	138
Использование строк в массивах	138
Риски, связанные с вводом строк	140
Построчное чтение ввода	141
Смешивание строкового и числового ввода	145
Введение в класс <code>string</code>	146
Инициализация строк в C++11	148
Другие формы строковых литералов	153
Введение в структуры	154
Использование структур в программах	155
Инициализация структур в C++11	158
Может ли структура содержать член типа <code>string</code> ?	158
Прочие свойства структур	158
Массивы структур	160
Битовые поля в структурах	161
Объединения	162
Перечисления	163
Установка значений перечислителей	165
Диапазоны значений перечислителей	165
Указатели и свободное хранилище	166
Объявление и инициализация указателей	169
Опасность, связанная с указателями	171
Указатели и числа	172
Выделение памяти с помощью операции <code>new</code>	172
Освобождение памяти с помощью операции <code>delete</code>	175
Использование операции <code>new</code> для создания динамических массивов	176
Указатели, массивы и арифметика указателей	179
Замечания по программе	180
Указатели и строки	184
Использование операции <code>new</code> для создания динамических структур	189
Автоматическое, статическое и динамическое хранилище	192
Комбинации типов	194
Альтернативы массивам	196
Шаблонный класс <code>vector</code>	196
Шаблонный класс <code>array</code> (C++11)	197
Сравнение массивов, объектов <code>vector</code> и объектов <code>array</code>	198
Резюме	199
Вопросы для самоконтроля	201
Упражнения по программированию	202
<b>Глава 5. Циклы и выражения отношений</b>	<b>205</b>
Введение в циклы <code>for</code>	206
Части цикла <code>for</code>	207
Возврат к циклу <code>for</code>	213
Изменение шага цикла	214
Доступ внутрь строк с помощью цикла <code>for</code>	215
Операции инкремента и декремента	216

Побочные эффекты и точки следования	217
Сравнение префиксной и постфиксной форм	218
Операции инкремента и декремента и указатели	219
Комбинация операций присваивания	220
Составные операторы, или блоки	221
Дополнительные синтаксические трюки — операция запятой	222
Выражения отношений	225
Присваивание, сравнение и вероятные ошибки	226
Сравнение строк в стиле C	228
Сравнение строк класса <code>string</code>	230
Цикл <code>while</code>	231
Замечания по программе	233
Сравнение циклов <code>for</code> и <code>while</code>	234
Построение цикла задержки	235
Цикл <code>do while</code>	237
Цикл <code>for</code> , основанный на диапазоне (C++11)	239
Циклы и текстовый ввод	240
Применение для ввода простого <code>cin</code>	240
Спасение в виде <code>cin.get(char)</code>	242
Выбор используемой версии <code>cin.get()</code>	243
Условие конца файла	243
Еще одна версия <code>cin.get()</code>	246
Вложенные циклы и двумерные массивы	250
Инициализация двумерного массива	251
Использование двумерного массива	252
Резюме	253
Вопросы для самоконтроля	254
Упражнения по программированию	255
<b>Глава 6. Операторы ветвления и логические операции</b>	257
Оператор <code>if</code>	258
Оператор <code>if else</code>	259
Форматирование операторов <code>if else</code>	261
Конструкция <code>if else if else</code>	262
Логические выражения	264
Логическая операция “ИЛИ”: <code>  </code>	264
Логическая операция “И”: <code>&amp;&amp;</code>	265
Логическая операция “НЕ”: <code>!</code>	270
Факты, связанные с логическими операциями	271
Альтернативные представления	272
Библиотека символьных функций <code>cctype</code>	272
Операция <code>?:</code>	275
Оператор <code>switch</code>	276
Использование перечислителей в качестве меток	279
Операторы <code>switch</code> и <code>if else</code>	280
Операторы <code>break</code> и <code>continue</code>	281
Замечания по программе	282
Циклы для чтения чисел	283
Замечания по программе	286

## 10 Содержание

Простой файловый ввод-вывод	287
Текстовый ввод-вывод и текстовые файлы	287
Запись в текстовый файл	288
Чтение текстового файла	292
Резюме	297
Вопросы для самоконтроля	298
Упражнения по программированию	300
<b>Глава 7. Функции как программные модули C++</b>	<b>303</b>
Обзор функций	304
Определение функции	305
Прототипирование и вызов функции	307
Аргументы функций и передача по значению	310
Множественные аргументы	312
Еще одна функция с двумя аргументами	314
Функции и массивы	316
Как указатели позволяют функциям обрабатывать массивы	318
Последствия использования массивов в качестве аргументов	318
Дополнительные примеры функций для работы с массивами	321
Функции, работающие с диапазонами массивов	327
Указатели и <code>const</code>	329
Функции и двумерные массивы	332
Функции и строки в стиле C	333
Функции с аргументами – строками в стиле C	334
Функции, возвращающие строки в стиле C	335
Функции и структуры	337
Передача и возврат структур	337
Еще один пример использования функций со структурами	339
Передача адресов структур	344
Функции и объекты класса <code>string</code>	346
Функции и объекты <code>array</code>	347
Замечания по программе	349
Рекурсия	349
Рекурсия с одиночным рекурсивным вызовом	349
Рекурсия с множественными рекурсивными вызовами	351
Указатели на функции	352
Основы указателей на функции	353
Пример с указателем на функцию	355
Вариации на тему указателей на функции	356
Упрощение объявлений с помощью <code>typedef</code>	360
Резюме	361
Вопросы для самоконтроля	362
Упражнения по программированию	363
<b>Глава 8. Дополнительные сведения о функциях</b>	<b>367</b>
Встроенные функции C++	368
Ссылочные переменные	371
Создание ссылочных переменных	371
Ссылки как параметры функций	374

Свойства и особенности ссылок	377
Временные переменные, ссылочные аргументы и квалификатор <code>const</code>	379
Использование ссылок при работе со структурами	381
Использование ссылок на объект класса	388
Еще один урок ООП: объекты, наследование и ссылки	391
Когда целесообразно использовать ссылочные аргументы	394
Аргументы по умолчанию	395
Перегрузка функций	398
Пример перегрузки	400
Когда целесообразно использовать перегрузку функций	403
Шаблоны функций	404
Перегруженные шаблоны	407
Ограничения шаблонов	408
Явные специализации	409
Создание экземпляров и специализация	412
Какую версию функции выбирает компилятор?	414
Эволюция шаблонных функций	421
Резюме	424
Вопросы для самоконтроля	425
Упражнения по программированию	426
<b>Глава 9. Модели памяти и пространства имен</b>	<b>429</b>
Раздельная компиляция	430
Продолжительность хранения, область видимости и компоновка	435
Область видимости и связывание	436
Автоматическая продолжительность хранения	437
Переменные со статической продолжительностью хранения	442
Спецификаторы и классификаторы	452
Функции и связывание	455
Языковое связывание	456
Схемы хранения и динамическое выделение памяти	457
Инициализация с помощью операции <code>new</code>	458
Когда <code>new</code> дает сбой	458
<code>new</code> : операции, функции и заменяющие функции	458
Операция <code>new</code> с размещением	459
Пространства имен	463
Традиционные пространства имен C++	463
Новое средство пространств имен	465
Пример пространства имен	472
Пространства имен и будущее	475
Резюме	476
Вопросы для самоконтроля	477
Упражнения по программированию	479
<b>Глава 10. Объекты и классы</b>	<b>483</b>
Процедурное и объектно-ориентированное программирование	484
Абстракции и классы	485
Что такое тип?	486
Классы в C++	486

## 12 Содержание

Реализация функций-членов класса	492
Использование классов	496
Изменение реализации	498
Обзор ситуации на текущий момент	499
Конструкторы и деструкторы классов	500
Усовершенствование класса <code>Stock</code>	505
Обзор конструкторов и деструкторов	513
Изучение объектов: указатель <code>this</code>	514
Массив объектов	520
Область видимости класса	522
Абстрактные типы данных	526
Резюме	530
Вопросы для самоконтроля	531
Упражнения по программированию	531
<b>Глава 11. Работа с классами</b>	<b>535</b>
Перегрузка операций	537
Время в наших руках: разработка примера перегрузки операции	538
Добавление операции сложения	540
Ограничения перегрузки	543
Дополнительные перегруженные операции	545
Что такое друзья?	547
Создание друзей	549
Общий вид друга: перегрузка операции <code>&lt;&lt;</code>	550
Перегруженные операции: сравнение функций-членов и функций, не являющихся членами	556
Дополнительные сведения о перегрузке: класс <code>Vector</code>	557
Использование члена, хранящего состояние	564
Перегрузка арифметических операций для класса <code>Vector</code>	566
Комментарии к реализации	568
Использование класса <code>Vector</code> при решении задачи случайного блуждания	569
Автоматические преобразования и приведения типов в классах	572
Преобразования и друзья	583
Резюме	585
Вопросы для самоконтроля	587
Упражнения по программированию	587
<b>Глава 12. Классы и динамическое выделение памяти</b>	<b>591</b>
Динамическая память и классы	592
Простой пример и статические члены класса	592
Специальные функции-члены	601
Новый усовершенствованный класс <code>String</code>	609
О чем следует помнить при использовании операции <code>new</code> в конструкторах	619
Замечания о возвращаемых объектах	622
Использование указателей на объекты	625
Обзор технических приемов	634
Моделирование очереди	635
Класс <code>Queue</code>	636
Класс <code>Customer</code>	646

Моделирование работы банкомата	649
Резюме	653
Вопросы для самоконтроля	655
Упражнения по программированию	656
<b>Глава 13. Наследование классов</b>	659
Начало: простой базовый класс	661
Порождение класса	663
Конструкторы: варианты доступа	664
Использование производного класса	667
Особые отношения между производным и базовым классами	669
Наследование: отношение <i>является</i>	671
Полиморфное открытое наследование	673
Разработка классов Brass и BrassPlus	674
Статическое и динамическое связывание	685
Совместимость типов указателей и ссылок	685
Виртуальные функции-члены и динамическое связывание	687
Что следует знать о виртуальных методах	690
Управление доступом: <code>protected</code>	693
Абстрактные базовые классы	694
Применение концепции абстрактных базовых классов	697
Философия АБК	702
Наследование и динамическое выделение памяти	703
Случай 1: производный класс не использует операцию <code>new</code>	703
Случай 2: производный класс использует операцию <code>new</code>	704
Пример наследования с динамическим выделением памяти и дружественными функциями	706
Обзор структуры класса	711
Функции-члены, генерируемые компилятором	711
Другие соображения относительно методов класса	713
Соображения по поводу открытого наследования	716
Сводка функций классов	720
Резюме	721
Вопросы для самоконтроля	721
Упражнения по программированию	723
<b>Глава 14. Повторное использование кода в C++</b>	727
Классы с объектами-членами	728
Класс <code>valarray</code> : краткий обзор	729
Проект класса <code>Student</code>	730
Пример класса <code>Student</code>	731
Закрытое наследование	738
Новый вариант класса <code>Student</code>	738
Включение или закрытое наследование?	745
Защищенное наследование	746
Переопределение доступа с помощью <code>using</code>	746
Множественное наследование	748
Краткий обзор множественного наследования	764
Шаблоны классов	765



## 14 Содержание

Определение шаблона класса	766
Более внимательный взгляд на шаблонные классы	771
Пример шаблона массива и нетипизированные аргументы	776
Универсальность шаблонов	778
Специализации шаблона	781
Шаблоны-члены	784
Шаблонные классы и друзья	788
Резюме	795
Вопросы для самоконтроля	797
Упражнения по программированию	799

## Глава 15. Друзья, исключения и многое другое 805

Друзья	806
Дружественные классы	806
Дружественные функции-члены	811
Другие дружественные отношения	813
Вложенные классы	815
Вложенные классы и доступ	817
Вложение в шаблонах	818
Исключения	821
Вызов <code>abort()</code>	822
Возврат кода ошибки	823
Механизм исключений	825
Использование объектов в качестве исключений	827
Спецификации исключений в C++11	831
Раскручивание стека	832
Дополнительные свойства исключений	837
Класс <code>exception</code>	839
Исключения, классы и наследование	843
Потеря исключений	847
Предостережения относительно использования исключений	850
Динамическая идентификация типов	852
Для чего нужен механизм RTTI	852
Как работает механизм RTTI	852
Операция <code>dynamic_cast</code>	853
Операции приведения типов	860
Резюме	863
Вопросы для самоконтроля	864
Упражнения по программированию	865

## Глава 16. Класс `string` и стандартная библиотека шаблонов 867

Класс <code>string</code>	868
Создание объекта <code>string</code>	868
Ввод для класса <code>string</code>	872
Работа со строками	875
Другие возможности, предлагаемые классом <code>string</code>	880
Разновидности строк	881
Классы шаблонов интеллектуальных указателей	882
Использование интеллектуальных указателей	883

Соображения по поводу интеллектуальных указателей	886
Выбор интеллектуального указателя	890
Стандартная библиотека шаблонов (STL)	892
Класс шаблона <code>vector</code>	892
Что еще можно делать с помощью векторов	894
Дополнительные возможности векторов	899
Цикл <code>for</code> , основанный на диапазоне (C++11)	903
Обобщенное программирование	903
Предназначение итераторов	904
Виды итераторов	908
Иерархия итераторов	911
Концепции, уточнения и модели	912
Виды контейнеров	918
Ассоциативные контейнеры	929
Неупорядоченные ассоциативные контейнеры (C++11)	935
Функциональные объекты (функторы)	936
Концепции функторов	937
Предопределенные функторы	939
Адаптируемые функторы и функциональные адаптеры	941
Алгоритмы	943
Группы алгоритмов	944
Основные свойства алгоритмов	944
Библиотека STL и класс <code>string</code>	946
Сравнение функций и методов контейнеров	947
Использование STL	949
Шаблон <code>initializer_list</code> (C++11)	957
Замечания по программе	960
Резюме	960
Вопросы для самоконтроля	962
Упражнения по программированию	963
<b>Глава 17. Ввод, вывод и файлы</b>	967
Обзор ввода и вывода в C++	968
Потоки и буферы	969
Потоки, буферы и файл <code>iostream</code>	971
Перенаправление	973
Вывод с помощью <code>cout</code>	975
Перегруженная операция <code>&lt;&lt;</code>	975
Другие методы <code>ostream</code>	977
Очистка выходного буфера	980
Форматирование с помощью <code>cout</code>	981
Ввод с помощью <code>cin</code>	996
Восприятие ввода операцией <code>cin &gt;&gt;</code>	998
Состояния потока	1000
Другие методы класса <code>istream</code>	1004
Другие методы класса <code>istream</code>	1011
Файловый ввод и вывод	1015
Простой файловый ввод-вывод	1016
Проверка потока и <code>is_open()</code>	1019

## 16 Содержание

Открытие нескольких файлов	1020
Обработка командной строки	1020
Режимы файла	1022
Произвольный доступ	1032
Внутреннее форматирование	1040
Резюме	1042
Вопросы для самопроверки	1044
Упражнения по программированию	1045
<b>Глава 18. Новый стандарт C++</b>	<b>1049</b>
Обзор уже известных функциональных средств C++11	1050
Новые типы	1050
Унифицированная инициализация	1050
Объявления	1052
<code>nullptr</code>	1054
Интеллектуальные указатели	1054
Изменения в спецификации исключений	1054
Перечисления с областью видимости	1054
Изменения в классах	1055
Изменения в шаблонах и STL	1056
Ссылка <code>rvalue</code>	1058
Семантика переноса и ссылка <code>rvalue</code>	1059
Необходимость в семантике переноса	1059
Пример семантики переноса	1061
Исследование конструктора переноса	1065
Присваивание	1067
Принудительное применение переноса	1067
Новые возможности классов	1071
Специальные функции-члены	1071
Явно заданные по умолчанию и удаленные методы	1072
Делегирование конструкторов	1074
Наследование конструкторов	1074
Управление виртуальными методами: <code>override</code> и <code>final</code>	1076
Лямбда-функции	1077
Как работают указатели на функции, функторы и лямбда	1077
Более подробно о лямбда-функциях	1081
Оболочки	1083
Оболочка <code>function</code> и неэффективность шаблонов	1084
Решение проблемы	1086
Дополнительные возможности	1087
Шаблоны с переменным числом аргументов	1088
Пакеты параметров шаблонов и функций	1089
Распаковка пакетов	1090
Использование рекурсии в шаблонных функциях с переменным числом аргументов	1090
Другие средства C++11	1093
Параллельное программирование	1093
Библиотечные дополнения	1094
Низкоуровневое программирование	1094

Смешанные средства	1095
Языковые изменения	1096
Проект Boost	1096
Проект TR1	1097
Использование Boost	1097
Что дальше?	1098
Резюме	1099
Вопросы для самоконтроля	1100
Упражнения по программированию	1102
<b>Приложение А. Основания систем счисления</b>	1105
Десятичные числа (основание 10)	1105
Восьмеричные целые числа (основание 8)	1106
Шестнадцатеричные числа (основание 16)	1106
Двоичные числа (основание 2)	1107
Двоичная и шестнадцатеричная формы записи	1107
<b>Приложение Б. Зарезервированные слова C++</b>	1109
Ключевые слова C++	1109
Альтернативные лексемы	1110
Зарезервированные имена библиотеки C++	1110
Идентификаторы со специальным назначением	1111
<b>Приложение В. Набор символов ASCII</b>	1113
<b>Приложение Г. Приоритеты операций</b>	1119
<b>Приложение Д. Другие операции</b>	1123
Битовые операции	1123
Операции сдвига	1123
Логические битовые операции	1125
Альтернативные представления битовых операций	1127
Примеры использования битовых операций	1128
Операции разыменования членов	1129
alignof (C++11)	1133
noexcept (C++11)	1134
<b>Приложение Е. Шаблонный класс string</b>	1135
Тринадцать типов и константа	1136
Информация о данных, конструкторы и вспомогательные элементы	1137
Конструктор по умолчанию	1139
Конструкторы, использующие строки в стиле C	1140
Конструкторы, использующие часть строки в стиле C	1140
Конструкторы, использующие ссылку lvalue	1140
Конструкторы, использующие ссылку rvalue (C++11)	1142
Конструктор, использующий n копий символа	1142
Конструктор, использующий диапазон	1142
Конструктор, использующий список инициализаторов (C++11)	1143
Различные действия с памятью	1143

## 18 Содержание

Доступ к строке	1143
Базовое присваивание	1145
Поиск в строках	1145
Семейство <code>find()</code>	1145
Семейство <code>rfind()</code>	1146
Семейство <code>find_first_of()</code>	1147
Семейство <code>find_last_of()</code>	1147
Семейство <code>find_first_not_of()</code>	1147
Семейство <code>find_last_not_of()</code>	1148
Методы и функции сравнения	1148
Модификация строк	1149
Методы присоединения и добавления	1150
Дополнительные методы присваивания	1150
Методы вставки	1151
Методы очистки	1152
Методы замены	1152
Другие методы модификации: <code>copy()</code> и <code>swap()</code>	1153
Ввод и вывод	1153
<b>Приложение Ж. Методы и функции стандартной библиотеки шаблонов</b>	<b>1155</b>
Библиотека STL и C++11	1155
Новые контейнеры	1155
Изменения в контейнерах C++98	1156
Члены, общие для всех или большинства контейнеров	1157
Дополнительные члены для контейнеров последовательностей	1160
Дополнительные операции для множеств и карт	1164
Неупорядоченные ассоциативные контейнеры (C++11)	1166
Функции библиотеки STL	1168
Операции, не модифицирующие последовательности	1169
Операции, видоизменяющие последовательности	1174
Операции сортировки и связанные с ними операции	1184
Числовые операции	1197
<b>Приложение З. Рекомендуемая литература и ресурсы в Интернете</b>	<b>1201</b>
Рекомендуемая литература	1201
Ресурсы в Интернете	1203
<b>Приложение И. Переход к стандарту ANSI/ISO C++</b>	<b>1205</b>
Используйте альтернативы для некоторых директив препроцессора	1205
Используйте <code>const</code> вместо <code>#define</code> для определения констант	1205
Используйте <code>inline</code> вместо <code>#define</code> для определения коротких функций	1207
Используйте прототипы функций	1208
Используйте приведения типов	1208
Знакомьтесь с функциональными возможностями C++	1209
Используйте новую организацию заголовочных файлов	1209
Используйте пространства имен	1209
Используйте интеллектуальные указатели	1211
Используйте класс <code>string</code>	1211
Используйте библиотеку STL	1211

<b>Приложение К. Ответы на вопросы для самоконтроля</b>	1213
Глава 2	1213
Глава 3	1214
Глава 4	1215
Глава 5	1217
Глава 6	1218
Глава 7	1219
Глава 8	1221
Глава 9	1223
Глава 10	1225
Глава 11	1227
Глава 12	1228
Глава 13	1230
Глава 14	1232
Глава 15	1233
Глава 16	1234
Глава 17	1236
Глава 18	1237
<b>Предметный указатель</b>	1240

# 9

## Модели памяти и пространства имен

### **В ЭТОЙ ГЛАВЕ...**

- Раздельная компиляция программ
- Продолжительность хранения, область видимости и компоновка
- Операция `new` с размещением
- Пространства имен

Язык C++ предлагает множество способов хранения данных в памяти. Имеется возможность выбора длительности хранения данных в памяти (продолжительность существования области хранения) и определения частей программы, имеющих доступ к данным (область видимости и связывание). Операция `new` позволяет динамически выделять память, а операция `new` с размещением является ее вариацией. Возможности пространства имен C++ обеспечивают дополнительный контроль над доступом к данным. Крупные программы обычно состоят из нескольких файлов исходного кода, которые могут совместно использовать определенные данные. Поскольку в таких программах применяется раздельная компиляция файлов, эта глава начинается с освещения данной темы.

## Раздельная компиляция

Язык C++, как и C, позволяет и даже поощряет размещение функций программы в отдельных файлах. Как говорилось в главе 1, файлы можно компилировать раздельно, а затем связывать их с конечным продуктом — исполняемой программой. (Как правило, компилятор C++ не только компилирует программы, но и управляет работой компоновщика.) При изменении только одного файла можно перекомпилировать лишь этот файл и затем связать его с ранее скомпилированными версиями других файлов. Этот механизм облегчает работу с крупными программами. Более того, большинство сред программирования на C++ предоставляют дополнительные средства, упрощающие такое управление. Например, в системах Unix и Linux имеется программа `make`, хранящая сведения обо всех файлах, от которых зависит программа, и о времени их последней модификации. После запуска `make` обнаруживает изменения в исходных файлах с момента последней компиляции, а затем предлагает выполнить соответствующие действия, необходимые для воссоздания программы. Большинство интегрированных сред разработки (*integrated development environment* — IDE), включая `Embarcadero C++ Builder`, `Microsoft Visual C++`, `Apple Xcode` и `Freescale CodeWarrior`, предоставляют аналогичные средства, доступ к которым осуществляется с помощью меню `Project` (Проект).

Рассмотрим простой пример. Вместо того чтобы разбирать детали компиляции, которые зависят от реализации, давайте сосредоточим внимание на более общих аспектах, таких как проектирование.

Предположим, что решено разделить программу из листинга 7.12 на части и поместить используемые ею функции в отдельный файл. Напомним, что эта программа преобразует прямоугольные координаты в полярные, после чего отображает результат. Нельзя просто вырезать из исходного файла часть кода после окончания функции `main()`. Дело в том, что `main()` и другие две функции используют одни и те же объявления структур, поэтому необходимо поместить эти объявления в оба файла. При простом наборе объявлений в коде можно допустить ошибку. Но даже если объявления скопированы безошибочно, при последующих модификациях нужно будет не забыть внести изменения в оба файла. Одним словом, разделение программы на несколько файлов создает новые проблемы.

Кому нужны дополнительные сложности? Только не разработчикам C и C++. Для решения подобных проблем была предоставлена директива `#include`. Вместо того чтобы помещать объявления структур в каждый файл, их можно разместить в заголовочном файле, а затем включать его в каждый файл исходного кода. Таким образом, изменения в объявление структуры будут вноситься только один раз в заголовочный файл. Кроме того, в заголовочный файл можно помещать прототипы функций.



Итак, исходную программу можно разбить на три части:

- заголовочный файл, содержащий объявления структур и прототипы функций, которые используют эти структуры;
- файл исходного кода, содержащий код функций, которые работают со структурами;
- файл исходного кода, содержащий код, который вызывает функции работы со структурами.

Такая стратегия может успешно применяться для организации программы. Если, например, создается другая программа, которая пользуется теми же самыми функциями, достаточно включить в нее заголовочный файл и добавить файл с функциями в проект или список make. К тому же такая организация программы соответствует принципам объектно-ориентированного программирования (ООП). Первый файл – заголовочный – содержит определения пользовательских типов. Второй файл содержит код функций для манипулирования типами, определенными пользователем. Вместе оба файла формируют пакет, который можно использовать в различных программах.

В заголовочный файл не следует помещать определения функций или объявления переменных. Хотя в простейших проектах такой подход может работать, обычно он приводит к проблемам. Например, если в заголовочном файле содержится определение функции, и этот заголовочный файл включен в два других файла, которые являются частью одной программы, в этой программе окажется два определения одной и той же функции, что вызовет ошибку, если только функция не является встроенной. В заголовочных файлах обычно содержится следующее:

- прототипы функций;
- символические константы, определенные с использованием `#define` или `const`;
- объявления структур;
- объявления классов;
- объявления шаблонов;
- встроенные функции.

Объявления структур можно помещать в заголовочные файлы, поскольку они не создают переменные, а только указывают компилятору, как создавать структурную переменную, когда она объявляется в файле исходного кода. Подобно этому объявления шаблонов – это не код, который нужно компилировать, а инструкции для компилятора, указывающие, каким образом генерировать определения функций, чтобы они соответствовали вызовам функций, встречающимся в исходном коде. Данные, объявленные как `const`, и встроенные функции имеют специальные свойства связывания (вскоре они будут рассмотрены), которые позволяют размещать их в заголовочных файлах, не вызывая при этом каких-либо проблем.

В листингах 9.1, 9.2 и 9.3 показан результат разделения программы из листинга 7.12 на отдельные части. Обратите внимание, что при включении заголовочного файла используется запись `"coordin.h"`, а не `<coordin.h>`. Если имя файла помещено в угловые скобки, компилятор C++ ищет его в той части базовой файловой системы, где расположены стандартные заголовочные файлы. Но когда имя файла представлено в двойных кавычках, компилятор сначала ищет файл в текущем рабочем каталоге или в каталоге с исходным кодом (либо в другом аналогичном месте, которое зависит от версии компилятора). Не обнаружив заголовочный файл там, он ищет его в стандартном местоположении. Таким образом, при включении собственных заголовочных файлов должны использоваться двойные кавычки, а не угловые скобки.

На рис. 9.1 показаны шаги по сборке этой программы в системе Unix. Обратите внимание, что пользователь только выдает команду компиляции `CC`, а остальные действия выполняются автоматически. Компиляторы командной строки `g++`, `gpp` и Borland C++ (`bcc32.exe`) ведут себя аналогичным образом. Среды разработки Apple Xcode, Embarcadero C++ Builder и Microsoft Visual C++ в сущности выполняют те же самые действия, однако, как упоминалось в главе 1, процесс инициируется по-другому, с помощью команд меню, которые позволяют создавать проект и ассоциировать с ним файлы исходного кода, но не заголовочные файлы. Дело в том, что заголовочными файлами управляет директива `#include`. Кроме того, не следует использовать директиву `#include` для включения файлов исходного кода, поскольку это может привести к дублированным объявлениям.

### Внимание!

В интегрированных средах разработки не добавляйте заголовочные файлы в список проекта и не используйте директиву `#include` для включения одних файлов исходного кода в другие файлы исходного кода.

1. Ввод команды компиляции для двух файлов исходного кода: `CC file1.cpp file2.cpp`
2. Препроцессор объединяет включенные файлы с исходным кодом:

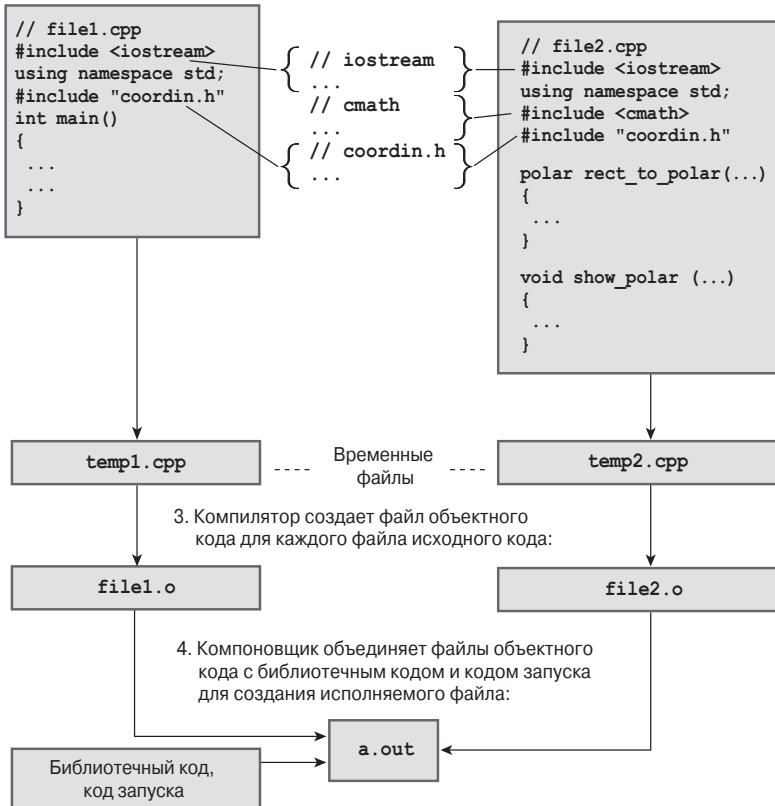


Рис. 9.1. Компиляция многофайловых программ C++ в системе Unix

**Листинг 9.1. coordin.h**


---

```
// coordin.h -- шаблоны структур и прототипы функций
// шаблоны структур
#ifndef COORDIN_H_
#define COORDIN_H_

struct polar
{
    double distance;    // расстояние от исходной точки
    double angle;      // направление от исходной точки
};

struct rect
{
    double x;          // расстояние по горизонтали от исходной точки
    double y;          // расстояние по вертикали от исходной точки
};

// прототипы
polar rect_to_polar(rect r);
void show_polar(polar p);

#endif
```

---

**Управление заголовочными файлами**

Заголовочный файл должен включаться в файл только один раз. Это кажется простым требованием, которое легко запомнить и придерживаться, тем не менее, можно непреднамеренно включить заголовочный файл несколько раз, даже не подозревая об этом. Например, предположим, что используется заголовочный файл, который включает другой заголовочный файл. В C/C++ существует стандартный прием, позволяющий избежать многократных включений заголовочных файлов. Он основан на использовании директивы препроцессора `#ifndef` (*if not defined* — если не определено). Показанный ниже фрагмент кода обеспечивает обработку операторов, находящихся между директивами `#ifndef` и `#endif`, только в случае, если имя `COORDIN_H_` не было определено ранее с помощью директивы препроцессора `#define`:

```
#ifndef COORDIN_H_
...
#endif
```

Обычно директива `#define` используется для создания символических констант, как в следующем примере:

```
#define MAXIMUM 4096
```

Однако для определения имени достаточно просто указать директиву `#define` с этим именем:

```
#define COORDIN_H_
```

Прием, применяемый в листинге 9.1, предусматривает помещение содержимого файла *внутри* `#ifndef`:

```
#ifndef COORDIN_H_
#define COORDIN_H_
// здесь размещается содержимое включаемого файла
#endif
```

Когда компилятор впервые сталкивается с этим файлом, имя `COORDIN_H_` должно быть неопределенным. (Во избежание совпадения с существующими именами, имя строится на основе имени включаемого файла и нескольких символов подчеркивания.)

В этом случае компилятор будет обрабатывать код между директивами `#ifndef` и `#endif`, что, собственно, и требуется. Во время обработки компилятор читает строку с директивой, определяющей имя `COORDIN_H_`. Если затем компилятор обнаруживает второе включение `coordin.h` в том же самом файле, он замечает, что имя `COORDIN_H_` уже определено, и переходит к строке, следующей после `#endif`. Обратите внимание, что данный прием не предотвращает повторного включения файла. Вместо этого он заставляет компилятор игнорировать содержимое всех включений кроме первого. Такая методика защиты используется в большинстве стандартных заголовочных файлов C и C++. Если ее не применять, одна и та же структура, например, окажется объявленной в файле дважды, что приведет к ошибке компиляции.

### Листинг 9.2. file1.cpp

---

```
// file1.cpp -- пример программы, состоящей из трех файлов
#include <iostream>
#include "coordin.h" // шаблоны структур, прототипы функций
using namespace std;
int main()
{
    rect rplace;
    polar pplace;

    cout << "Enter the x and y values: ";           // ввод значений x и y
    while (cin >> rplace.x >> rplace.y)           // ловкое использование cin
    {
        pplace = rect_to_polar(rplace);
        show_polar(pplace);
        cout << "Next two numbers (q to quit): ";
            // ввод следующих двух чисел (q для завершения)
    }
    cout << "Done.\n";
    return 0;
}
```

---

### Листинг 9.3. file2.cpp

---

```
// file2.cpp -- содержит функции, вызываемые в file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h" // шаблоны структур, прототипы функций

// Преобразование прямоугольных координат в полярные
polar rect_to_polar(rect rpos)
{
    using namespace std;
    polar answer;
    answer.distance =
        sqrt( rpos.x * rpos.x + rpos.y * rpos.y );
    answer.angle = atan2(rpos.y, rpos.x);
    return answer; // возврат структуры polar
}

// Отображение полярных координат с преобразованием радиан в градусы
void show_polar (polar dapos)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;
    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
}
```

---

В результате компиляции и компоновки этих двух файлов исходного кода и нового заголовочного файла получается исполняемая программа. Ниже приведен пример ее выполнения:

```
Enter the x and y values: 120 80
distance = 144.222, angle = 33.6901 degrees
Next two numbers (q to quit): 120 50
distance = 130, angle = 22.6199 degrees
Next two numbers (q to quit): q
```

Кстати, хотя мы обсудили отдельную компиляцию в терминах файлов, в стандарте C++ вместо термина *файл* используется термин *единица трансляции*, чтобы сохранить более высокую степень обобщенности; файловая модель — это не единственный способ организации информации в компьютере. Для простоты в этой книге будет применяться термин “файл”, но помните, что под этим понимается также и “единица трансляции”.

### Связывание с множеством библиотек

Стандарт C++ предоставляет каждому разработчику компилятора возможность самостоятельной реализации декорирования имен (см. врезку “Что такое декорирование имен?” в главе 8), поэтому следует учитывать, что связывание двоичных модулей (файлов объектного кода), созданных различными компиляторами, скорее всего, не будет успешным. Другими словами, для одной и той же функции два компилятора сгенерируют различные декорированные имена. Такое различие в именах не позволит компоновщику найти соответствие между вызовом функции, сгенерированной одним компилятором, и определением функции, сгенерированной другим компилятором. Перед компоновкой скомпилированных модулей нужно обеспечить, чтобы каждый объектный файл или библиотека была сгенерирована одним и тем же компилятором. При наличии исходного кода проблемы компоновки обычно легко решаются за счет повторной компиляции.

## Продолжительность хранения, область видимости и компоновка

После обзора многофайловых программ пришло время продолжить рассмотрение моделей памяти, начатое в главе 4. Дело в том, что категории хранения влияют на то, как информация может совместно использоваться разными файлами. Вспомните, что в главе 4 говорилось о памяти. В языке C++ применяются три различных схемы хранения данных (в C++11 их четыре). Эти схемы отличаются между собой продолжительностью нахождения данных в памяти.

- **Автоматическая продолжительность хранения.** Переменные, объявленные внутри определения функции — включая параметры функции — имеют автоматическую продолжительность хранения. Они создаются, когда выполнение программы входит в функцию или блок, где эти переменные определены. После выхода из блока или функции используемая переменными память освобождается. В C++ существуют два вида автоматических переменных.
- **Статическая продолжительность хранения.** Переменные, объявленные за пределами определения функции либо с использованием ключевого слова `static`, имеют статическую продолжительность хранения. Они существуют в течение всего времени выполнения программы. В языке C++ существуют три вида переменных со статической продолжительностью хранения.

- **Потоковая продолжительность хранения (C++11).** В наши дни многоядерные процессоры распространены практически повсеместно. Такие процессоры способны поддерживать множество выполняющихся задач одновременно. Это позволяет программе разделить вычисления на отдельные *потоки*, которые могут быть обработаны параллельно. Переменные, объявленные с ключевым словом `thread_local`, хранятся на протяжении времени существования содержащего их потока. Вопросы параллельного программирования в этой книге не рассматриваются.
- **Динамическая продолжительность хранения.** Память, выделяемая операцией `new`, сохраняется до тех пор, пока она не будет освобождена с помощью операции `delete` или до завершения программы, смотря какое из событий наступит раньше. Эта память имеет динамическую продолжительность хранения и часто называется *свободным хранилищем* или *кучей*.

Далее мы продолжим изучение понятий области видимости переменных (их доступности для программы) и компоновки, которая определяет, какая информация совместно используется разными файлами.

## Область видимости и связывание

*Область видимости* (или *контекст*) определяет доступность имени в пределах файла (единицы трансляции). Например, переменная, определенная в функции, может быть использована только в этой функции, но не в какой-либо другой, в то время как переменная, определенная в файле до определений функций, может применяться во всех функциях. *Связывание* описывает, как имя может разделяться различными единицами трансляции. Имя с *внешним связыванием* может совместно использоваться разными файлами, а имя с *внутренним связыванием* — функциями внутри одного файла. Имена автоматических переменных не имеют никакого связывания, поскольку они не являются разделяемыми.

Переменная C++ может иметь одну из нескольких возможных областей видимости. Переменная с *локальной областью видимости* (которая также называется *областью видимости блока*) известна только внутри блока, где она определена. Вспомните, что блок — это последовательность операторов, заключенная в фигурные скобки. Например, тело функции является блоком, однако в него могут быть вложены и другие блоки. Переменная, имеющая *глобальную область видимости* (которая часто называется *областью видимости файла*), известна во всем файле, начиная с точки, где она определена. Автоматические переменные имеют локальную область видимости, а статические переменные могут иметь различную область видимости в зависимости от того, как они определены. Имена, используемые в *области видимости прототипа функции*, доступны только в пределах круглых скобок, которые содержат список аргументов. (Вот почему не важно, что они собой представляют и присутствуют ли вообще.) Элементы, объявленные в классе, имеют *область видимости класса* (см. главу 10). Переменные, объявленные в пространстве имен, имеют *область видимости пространства имен*. (Теперь, когда в C++ были добавлены пространства имен, глобальная область видимости стала частным случаем области видимости пространства имен.)

Функции C++ могут иметь область видимости класса или область видимости пространства имен, включая глобальную область видимости, но не могут иметь локальную область видимости. (Функция не может быть определена внутри блока, поскольку если бы она могла иметь локальную область видимости, то была бы известна только самой себе и, следовательно, не могла быть вызванной из другой функции. Такая функция вообще не могла бы считаться функцией.)

Различные варианты хранения в C++ характеризуются продолжительностью существования, областью видимости и связыванием. Давайте рассмотрим классы хранения C++ в терминах их свойств. Начнем с исследования ситуации, имевшей место до ввода в язык пространств имен, и посмотрим, как они изменили общую картину.

## Автоматическая продолжительность хранения

Параметры функции и переменные, объявленные внутри функции, по умолчанию имеют автоматическую продолжительность хранения. Они также обладают локальной областью видимости и не имеют связывания. Другими словами, если объявить переменную по имени `texas` в `main()`, а затем объявить еще одну переменную с тем же именем в функции `oil()`, будут созданы две независимые переменные, каждая из которых известна только в той функции, в которой объявлена. Любые операции с переменной `texas` в функции `oil()` не оказывают влияния на переменную `texas` в `main()` и наоборот. Кроме того, каждой переменной выделяется память, когда выполнение программы входит в самый вложенный блок, содержащий определение переменной, и каждая переменная прекращает существование, когда выполнение программы покидает этот блок. (Обратите внимание, что переменной выделяется память, когда выполнение программы входит в такой блок, но область видимости начинается только после точки объявления.)

Если определить переменную внутри блока, ее время существования и область видимости ограничиваются этим блоком. Предположим, что в начале `main()` определена переменная `teledeli`. Теперь пусть в `main()` создается новый блок, в котором определяется новая переменная по имени `websight`. В этом случае переменная `teledeli` является видимой как во внешнем, так и во внутреннем блоке, в то время как `websight` существует только во внутреннем блоке и находится в области видимости с точки своего определения до тех пор, пока выполнение программы не доберется до конца блока:

```
int main()
{
    int teledeli = 5;
    { // Переменной websight выделяется память
        cout << "Hello\n";
        int websight = -2; // начинается область видимости websight
        cout << websight << ' ' << teledeli << endl;
    } // websight прекращает существование
    cout << teledeli << endl;
    ...
} // Переменная teledeli прекращает существование
```

А что если переменной во внутреннем блоке назначить имя `teledeli` вместо `websight`, в результате чего получится две переменных с одним и тем же именем, одна из которых находится во внешнем блоке, а другая — во внутреннем? В этом случае программа интерпретирует имя `teledeli` как переменную, локальную по отношению к блоку, во время выполнения операторов этого блока. Принято говорить, что новое определение *скрывает* предыдущее. Новое определение попадает в область видимости, а предыдущее из нее временно удаляется. Когда выполнение программы покидает блок, исходное определение возвращается обратно в область видимости (рис. 9.2).

Код в листинге 9.4 показывает, что автоматические переменные локализованы внутри функций или блоков, которые их содержат.

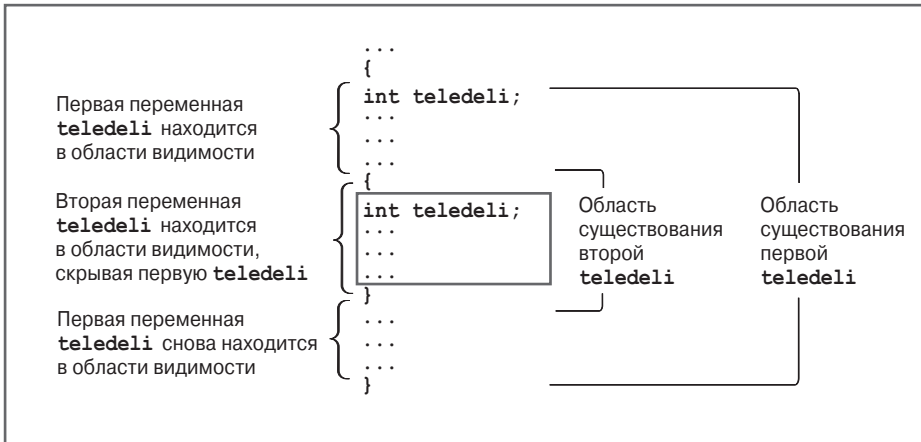


Рис. 9.2. Блоки и область видимости

## Листинг 9.4. auto.cpp

```
// auto.cpp -- иллюстрация области видимости автоматических переменных
#include <iostream>
void oil(int x);
int main()
{
    using namespace std;
    int texas = 31;
    int year = 2011;
    cout << "In main(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In main(), year = " << year << ", &year = ";
    cout << &year << endl;
    oil(texas);
    cout << "In main(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In main(), year = " << year << ", &year = ";
    cout << &year << endl;
    return 0;
}
void oil(int x)
{
    using namespace std;
    int texas = 5;
    cout << "In oil(), texas = " << texas << ", &texas = ";
    cout << &texas << endl;
    cout << "In oil(), x = " << x << ", &x = ";
    cout << &x << endl;
    {
        // начало блока
        int texas = 113;
        cout << "In block, texas = " << texas;
        cout << ", &texas = " << &texas << endl;
        cout << "In block, x = " << x << ", &x = ";
        cout << &x << endl;
    }
    // конец блока
    cout << "Post-block texas = " << texas;
    cout << ", &texas = " << &texas << endl;
}

```



Ниже показан вывод программы из листинга 9.4:

```
In main(), texas = 31, &texas = 0012FED4
In main(), year = 2011, &year = 0012FEC8
In oil(), texas = 5, &texas = 0012FDE4
In oil(), x = 31, &x = 0012FDF4
In block, texas = 113, &texas = 0012FDD8
In block, x = 31, &x = 0012FDF4
Post-block texas = 5, &texas = 0012FDE4
In main(), texas = 31, &texas = 0012FED4
In main(), year = 2011, &year = 0012FEC8
```

Обратите внимание, что каждая из трех переменных `texas` в листинге 9.4 имеет собственный, отличающийся от других адрес, и программа использует только ту переменную, которая в данное время находится в области видимости. Поэтому присваивание переменной `texas` значения 113 во внутреннем блоке функции `oil()` никак не отражается на других переменных с тем же именем. (Как обычно, конкретные значения адресов и формат представления варьируются от системы к системе.)

Рассмотрим последовательность событий. Когда `main()` начинается, программа выделяет память для переменных `texas` и `year`, и они обе попадают в область видимости. Когда программа вызывает функцию `oil()`, эти переменные остаются в памяти, но покидают область видимости. Две новых переменных, `x` и `texas`, размещаются в памяти и попадают в область видимости. Когда выполнение программы достигает внутреннего блока в функции `oil()`, новая переменная `texas` выходит из области видимости (скрывается), поскольку замещается более новым определением. Однако переменная `x` остается в области видимости, т.к. в блоке новая переменная с таким же именем не определяется. Когда выполнение программы выходит за пределы этого блока, освобождается память, занятая самой новой переменной `texas`, а вторая переменная `texas` возвращается в область видимости. После завершения функции `oil()` переменные `texas` и `x` перестают существовать, а в область видимости возвращаются исходные переменные `texas` и `year`.

### Изменения, связанные с ключевым словом `auto`, в C++11

В C++11 ключевое слово `auto` используется для автоматического выведения типа, как уже было показано в главах 3, 7 и 8. Однако в C и предшествующих версиях C++ ключевое слово `auto` имеет совершенно другое предназначение. Оно служит для явного указания того, что переменная имеет автоматическое хранение:

```
int froob(int n)
{
    auto float ford; // ford получает автоматическое хранение
    ...
}
```

Поскольку программисты могут использовать ключевое слово `auto` только с переменными, которые по умолчанию уже являются автоматическими, это слово применялось редко. Главное его назначение заключается в документировании того факта, что действительно требуется локальная автоматическая переменная.

В C++11 такое использование больше не является допустимым. Люди, занимающиеся подготовкой стандартов, неохотно изменяют назначение ключевых слов, потому что в результате может перестать работать код, в котором эти ключевые слова применялись для других целей. В данном случае было высказано мнение, что в прошлом слово `auto` использовалось настолько редко, что вполне допустимо перепрофилировать его, нежели вводить новое ключевое слово.

### Инициализация автоматических переменных

Автоматическую переменную можно инициализировать с помощью любого выражения, значение которого известно на момент объявления переменной. Ниже приведен пример инициализации переменных `x`, `big`, `y` и `z`:

```
int w; // значение w не определено
int x = 5; // инициализация числовым литералом
int big = INT_MAX - 1; // инициализация константным выражением
int y = 2 * x; // использование ранее определенного значения x
cin >> w;
int z = 3 * w; // использование нового значения w
```

### Автоматические переменные и стек

Чтобы получить более полное представление об автоматических переменных, рассмотрим их реализацию обычным компилятором C++. Поскольку количество автоматических переменных растет или сокращается по мере того, как функции начинают и завершают выполнение, программа должна управлять автоматическими переменными в процессе своей работы. Стандартная методика состоит в выделении области памяти, которая будет использоваться в качестве стека, управляющего движением переменных.

Термин *стек* применяется потому, что новые данные размещаются, образно говоря, поверх старых данных (т.е. в смежных, а не в тех же самых ячейках памяти), а затем удаляются из стека, после того как программа завершит работу с ними. По умолчанию размер стека зависит от реализации, однако обычно компилятор предоставляет опцию изменения размера стека.

Программа отслеживает состояние стека с помощью двух указателей. Один указывает на базу стека, с которой начинается выделенная область памяти, а другой — на вершину стека, которая представляет собой следующую ячейку свободной памяти. Когда происходит вызов функции, ее автоматические переменные добавляются в стек, а указатель вершины устанавливается на свободную ячейку памяти, следующую за только что размещенными переменными. После завершения функции указатель вершины снова принимает значение, которое он имел до вызова функции. В результате эффективно освобождается память, которая использовалась для хранения новых переменных.

Стек построен по принципу LIFO (last-in, first-out — последним пришел, первым обслужен). Это означает, что переменная, которая попала в стек последней, удаляется из него первой. Такой механизм упрощает передачу аргументов. Вызов функции помещает значения ее аргументов в вершину стека и переустанавливает указатель вершины. Вызванная функция использует описание своих формальных параметров для определения адреса каждого аргумента. Например, на рис. 9.3 показана функция `fib()`, которая в момент вызова передает двухбайтное значение типа `int` и четырехбайтное значение типа `long`. Эти значения помещаются в стек. Когда функция `fib()` начинает выполняться, она связывает имена `real` и `tell` с этими двумя значениями. После завершения работы функции `fib()` указатель вершины стека возвращается в прежнее состояние. Новые значения не удаляются, но теперь они лишаются меток, и пространство памяти, которое они занимают, будет использовано следующим процессом, в ходе которого будут размещаться значения в стеке. (На рис. 9.3 показана упрощенная картина, поскольку при вызове функции может передаваться дополнительная информация, такая как адрес возврата.)

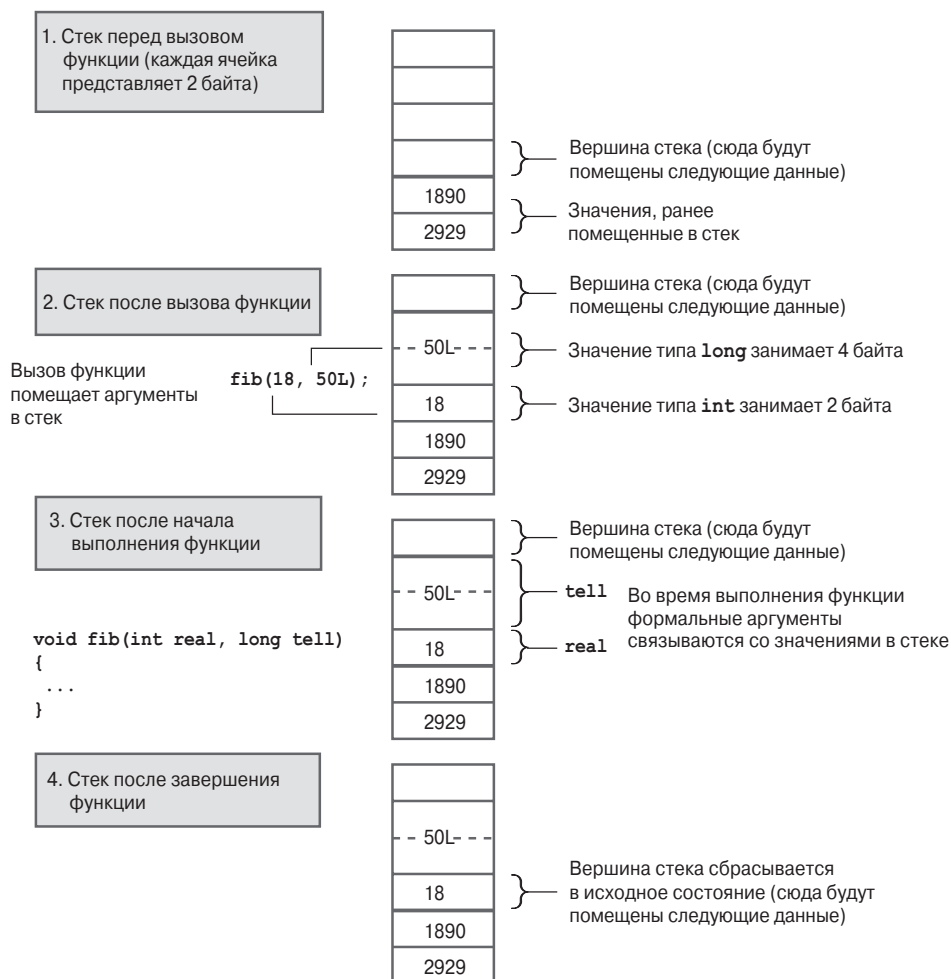


Рис. 9.3. Передача аргументов с использованием стека

### Регистровые переменные

Ключевое слово `register` было первоначально введено в языке C, чтобы рекомендовать компилятору использовать для хранения автоматической переменной регистр центрального процессора:

```
register int count_fast; // запрос на создание регистровой переменной
```

Идея заключалась в том, что это ускоряло доступ к переменной.

До появления стандарта C++11 это ключевое слово применялось в C++ похожим образом, но с одним отличием. Поскольку оборудование и компиляторы стали более совершенными, эта рекомендация была обобщена и начала указывать на тот факт, что переменная интенсивно используется, и, возможно, компилятор сумеет уделить ей особое внимание. В C++11 эта рекомендация является устаревшей и ключевое слово `register` остается просто способом идентифицировать переменную как автоматическую. Учитывая, что `register` может применяться только с переменными, которые будут автоматическими в любом случае, одна из причин использования этого ключево-

го слова — указать, что действительно нужна автоматическая переменная, возможно, с тем же самым именем, что и у внешней переменной. Точно таким же было первоначальное назначение `auto`. Однако более важная причина того, что ключевое слово `register` осталось, связана с желанием сохранить допустимым существующий код, в котором оно используется.

## Переменные со статической продолжительностью хранения

Язык C++, как и C, предоставляет переменные со статической продолжительностью хранения с тремя видами связывания: внешнее (возможность доступа в разных файлах), внутреннее (возможность доступа к функциям внутри одного файла) и отсутствие связывания (возможность доступа только к одной функции или к одному блоку внутри функции). Переменные с этими тремя типами связывания существуют в течение всего времени выполнения программы; они долговечнее автоматических переменных. Поскольку количество статических переменных не меняется на протяжении выполнения программы, она не нуждается в специальных механизмах, подобных стеку, чтобы управлять ими. Компилятор просто резервирует фиксированный блок памяти для хранения всех статических переменных, и эти переменные доступны программе на протяжении всего времени ее выполнения. Более того, если статическая переменная не инициализирована явно, компилятор устанавливает ее в 0. Элементы статических массивов и структур устанавливаются в 0 по умолчанию.

### На заметку!

Классический стандарт K&R C не позволяет инициализировать автоматические массивы и структуры, но допускает инициализацию статических массивов и структур. В ANSI C и C++ разрешено инициализировать обе разновидности данных. Однако некоторые ранние трансляторы C++ используют компиляторы языка C, которые не полностью совместимы со стандартом ANSI C. Если вы пользуетесь такой реализацией, то для инициализации массивов и структур может возникнуть необходимость воспользоваться одним из трех видов статических классов хранения.

Рассмотрим создание всех трех видов переменных со статической продолжительностью хранения, а затем приступим к исследованию их свойств. Чтобы создать статическую переменную с внешним связыванием, ее нужно объявить вне всех блоков. Чтобы создать статическую переменную с внутренним связыванием, ее следует объявить вне всех блоков и указать модификатор класса хранения `static`. Для создания статической переменной без связывания ее нужно объявить внутри какого-либо блока, используя модификатор `static`. В следующем фрагменте кода демонстрируются все три случая:

```
...
int global = 1000;           // статическая продолжительность, внешнее связывание
static int one_file = 50;   // статическая продолжительность, внутреннее связывание
int main()
{
    ...
}

void funct1(int n)
{
    static int count = 0;    // статическая продолжительность, нет связывания
    int llama = 0;
    ...
}
```

```
void funct2(int q)
{
    ...
}
```

Как уже упоминалось ранее, все переменные со статической продолжительностью хранения (в приведенном примере `global`, `one_file` и `count`) существуют с момента начала выполнения программы и до ее завершения. Переменная `count`, объявленная внутри функции `funct1()`, характеризуется локальной областью видимости и отсутствием связывания. Это означает, что она может использоваться только в рамках функции `funct1()`, точно так же как автоматическая переменная `llama`. Но, в отличие от `llama`, переменная `count` остается в памяти, даже когда функция `funct1()` не выполняется. Переменные `global` и `one_file` имеют область видимости файла, а это значит, что они могут использоваться, начиная с точки объявления и до конца файла. В частности, с обеими переменными можно работать в функциях `main()`, `funct1()` и `funct2()`. Так как переменная `one_file` имеет внутреннее связывание, она может использоваться только в файле, содержащем этот код. Поскольку переменная `global` имеет внешнее связывание, она также может применяться в других файлах, которые являются частью программы.

Все статические переменные обладают следующей особенностью инициализации: все биты неинициализированной статической переменной устанавливаются в 0. Такая переменная называется *инициализированной нулями*.

В табл. 9.1 приведена сводка по характеристикам классов хранения, которые были актуальны до появления пространств имен. Далее мы рассмотрим разновидности переменных со статической продолжительностью хранения более подробно.

**Таблица 9.1. Пять видов хранения переменных**

Описание хранения	Продолжительность	Область видимости	Связывание	Способ объявления
Автоматическая	Автоматическая	Блок	Нет	В блоке
Регистровая	Автоматическая	Блок	Нет	В блоке, с использованием ключевого слова <code>register</code>
Статическая без связывания	Статическая	Блок	Нет	В блоке, с использованием ключевого слова <code>static</code>
Статическая с внешним связыванием	Статическая	Файл	Внешнее	Вне всех функций
Статическая с внутренним связыванием	Статическая	Файл	Внутреннее	Вне всех функций, с использованием ключевого слова <code>static</code>

Обратите внимание, что в двух случаях упоминания в табл. 9.1 ключевое слово `static` имеет несколько отличающийся смысл. При использовании в локальном объявлении для указания статической переменной без связывания `static` отражает вид продолжительности хранения. Когда ключевое слово `static` применяется с объявлением вне блока, оно отражает внутреннее связывание; переменная уже имеет статическую продолжительность хранения. Это можно назвать *перегрузкой ключевого слова*, причем более точный смысл определяется контекстом.

### Инициализация статических переменных

Статические переменные могут быть инициализированными нулями, они могут быть подвергнуты *инициализации константным выражением*, и они могут быть подвергнуты *динамической инициализации*. Как вы уже, наверное, догадались, инициализация нулями означает установку переменной в значение ноль. Для скалярных типов ноль предусматривает приведение к соответствующему типу. Например, нулевой указатель, который представлен как 0 в коде C++, может иметь ненулевое внутреннее представление, поэтому переменная типа указателя будет инициализирована этим значением. Члены структуры являются инициализированными нулями, и любой заполняющий бит установлен в ноль.

Инициализация нулями и инициализация константным выражением вместе называются *статической инициализацией*. Это значит, что переменная инициализируется, когда компилятор обрабатывает файл (или единицу трансляции). Динамическая инициализация означает, что переменная инициализируется позже.

Так что же определяет, какая форма инициализации будет применена? Прежде всего, все статические переменные являются инициализированными нулями, указана какая-либо инициализация или нет. Далее, если переменная инициализируется константным выражением, которое компилятор может вычислить исключительно на основе содержимого файла (учитывая включаемые заголовочные файлы), возможно проведение инициализации константным выражением. При необходимости компилятор готов выполнить простые вычисления. Если к этому моменту информации недостаточно, переменная будет инициализирована динамически.

Рассмотрим следующий код:

```
#include <cmath>
int x; // инициализация нулями
int y = 5; // инициализация константным выражением
long z = 13 * 13; // инициализация константным выражением
const double pi = 4.0 * atan(1.0); // динамическая инициализация
```

В начале переменные `x`, `y`, `z` и `pi` являются инициализированными нулями. Затем компилятор вычисляет константные выражения и инициализирует `y` и `z`, соответственно, значениями 5 и 169. Но инициализация `pi` требует вызова функции `atan()`, и это должно подождать до тех пор, пока функция не будет скомпонована, а программа запущена.

Константное выражение не ограничено арифметическими выражениями, использующими литеральные константы. Например, в нем можно применять операцию `sizeof`:

```
int enough = 2 * sizeof(long) + 1; // инициализация константным выражением
```

В C++11 появилось новое ключевое слово `constexpr`, расширяющее возможности по созданию константных выражений; это одно из новых средств C++11, которые в настоящей книге не рассматриваются.

### Статическая продолжительность хранения, внешнее связывание

Переменные с внешним связыванием часто называются просто *внешними переменными*. Они обязательно имеют статическую продолжительность хранения и область видимости файла. Внешние переменные определяются вне всех функций и поэтому являются внешними по отношению к любой функции. Например, они могут быть объявлены до описания функции `main()` или в заголовочном файле. Внешнюю переменную можно использовать в любой функции, которая следует в файле после опреде-

ления переменной. Поэтому внешние переменные также называются *глобальными* — в отличие от автоматических переменных, которые являются локальными.

### Правило одного определения

С одной стороны, внешняя переменная должна быть объявлена в каждом файле, в котором она будет использоваться. С другой стороны, в C++ имеется так называемое “правило одного определения” (one definition rule — odr), которое гласит, что для каждой переменной должно существовать только одно определение. Чтобы удовлетворить этим требованиям, в C++ доступно два вида объявления переменных. Первый вид называется *определяющим объявлением* или просто *определением*. Определение приводит к выделению памяти для переменной. Второй вид называется *ссылочным объявлением* или просто *объявлением*. Объявление не приводит к выделению памяти, поскольку ссылается на переменную, которая уже существует.

Ссылочное объявление использует ключевое слово `extern` и не предоставляет возможности инициализации. В противном случае объявление является определением и приводит к выделению пространства для хранения:

```
double up;           // определение, up равно 0
extern int blem;     // переменная blem определена в другом месте
extern char gr = 'z'; // определение, поскольку присутствует инициализация
```

Если внешняя переменная используется в нескольких файлах, только один из них может содержать определение этой переменной (согласно правилу одного определения). Но во всех прочих файлах, где эта переменная используется, она должна быть объявлена с указанием ключевого слова `extern`:

```
// file01.cpp
extern int cats = 20; // определение, поскольку присутствует инициализация
int dogs = 22;       // тоже определение
int fleas;           // и это определение
...
// file02.cpp
// используются cats и dogs из file01.cpp
extern int cats;     // это не определения, поскольку в них указано
extern int dogs;     // ключевое слово extern и отсутствует инициализация
...
// file98.cpp
// используются cats, dogs и fleas из file01.cpp
extern int cats;
extern int dogs;
extern int fleas;
...
```

В этом случае во всех файлах используются переменные `cats` и `dogs`, определенные в `file01.cpp`. Однако в `file02.cpp` переменная `fleas` не объявляется повторно, поэтому доступ к ней невозможен. Ключевое слово `extern` в `file01.cpp` в действительности не нужно, поскольку и без него эффект будет таким же (рис. 9.4).

Обратите внимание, что правило одного определения не означает возможность существования только одной переменной с заданным именем. Например, автоматические переменные, разделяющие между собой одно и то же имя, но определенные в разных функциях, являются отдельными переменными, не зависящими друг от друга, и каждая из них обладает собственным адресом. Кроме того, как будет показано в последующих примерах, локальная переменная может скрывать глобальную переменную с тем же самым именем.

```
// программа file1.cpp
#include <iostream>
using namespace std;

// прототипы функций
#include "mystuff.h"

// определение внешней переменной
int process_status = 0;

void promise();
int main()
{
    ...
}

void promise()
{
    ...
}
```

В этом файле определяется переменная `process_status`, в результате чего компилятор выделяет для нее память.

```
// программа file2.cpp
#include <iostream>
using namespace std;

// прототипы функций
#include "mystuff.h"

// ссылка на внешнюю переменную
extern int process_status;

int manipulate(int n)
{
    ...
}

char * remark(char * str)
{
    ...
}
```

В этом файле используется ключевое слово `extern`, которое указывает программе использовать переменную `process_status`, определенную в другом файле.

*Рис. 9.4. Определяющее объявление и ссылочное объявление*

Тем не менее, хотя в программе могут присутствовать различные переменные с одинаковыми именами, каждая версия может иметь только одно определение.

А что если определить внешнюю переменную и затем объявить обычную переменную с тем же самым именем внутри функции? Второе объявление интерпретируется как определение автоматической переменной. Эта автоматическая переменная находится в области видимости, когда программа выполняет эту конкретную функцию. Код в листингах 9.5 и 9.6 в случае совместной компиляции иллюстрирует использование внешней переменной в двух файлах и сокрытие глобальной переменной объявлением автоматической переменной с тем же именем. Программа также демонстрирует применение ключевого слова `extern` для повторного объявления внешней переменной, определенной ранее, а также использование операции разрешения контекста для реализации доступа к иначе скрытой внешней переменной.

#### Листинг 9.5. `external.cpp`

```
// external.cpp -- внешние переменные
// Компилировать вместе с support.cpp
#include <iostream>
using namespace std;
// Внешняя переменная
double warming = 0.3; // переменная warming определена
// Прототипы функций
void update(double dt);
void local();
int main() // использует глобальную переменную
{
    cout << "Global warming is " << warming << " degrees.\n";
    update(0.1); // вызов функции, изменяющей warming
    cout << "Global warming is " << warming << " degrees.\n";
    local(); // вызов функции с локальной переменной warming
    cout << "Global warming is " << warming << " degrees.\n";
    return 0;
}
```



**Листинг 9.6. support.cpp**


---

```
// support.cpp -- использование внешних переменных
// Компилировать вместе с external.cpp
#include <iostream>
extern double warming;           // использование переменной warming из другого файла
// Прототипы функций
void update(double dt);
void local();

using std::cout;
void update(double dt)           // модифицирует глобальную переменную
{
    extern double warming;       // необязательное повторное объявление
    warming += dt;               // использование глобальной переменной warming
    cout << "Updating global warming to " << warming;
    cout << " degrees.\n";
}
void local()                     // использует локальную переменную
{
    double warming = 0.8;        // новая переменная скрывает внешнюю переменную
    cout << "Local warming = " << warming << " degrees.\n";
    // Доступ к глобальной переменной с помощью операции разрешения контекста
    cout << "But global warming = " << ::warming;
    cout << " degrees.\n";
}
}
```

---

Ниже показан вывод программы из листингов 9.5 и 9.6:

```
Global warming is 0.3 degrees.
Updating global warming to 0.4 degrees.
Global warming is 0.4 degrees.
Local warming = 0.8 degrees.
But global warming = 0.4 degrees.
Global warming is 0.4 degrees.
```

**Замечания по программе**

Вывод программы из листингов 9.5 и 9.6 показывает, что функции `main()` и `update()` имеют доступ к внешней переменной `warming`. Обратите внимание, что изменение, которое вносит функция `update()` в переменную `warming`, проявляется при последующих обращениях к этой переменной.

Определение переменной `warming` находится в листинге 9.5:

```
double warming = 0.3; // переменная warming определена
```

В листинге 9.6 применяется ключевое слово `extern`, чтобы сделать переменную `warming` доступной функциям из этого файла:

```
extern double warming; // использование переменной warming из другого файла
```

Это объявление означает: использовать переменную `warming`, определенную где-то во внешнем файле.

Вдобавок в функции `update()` осуществляется повторное объявление переменной `warming` за счет использования ключевого слова `extern`. Это ключевое слово означает: использовать переменную с таким именем, которая была внешне определена ранее. Поскольку функция `update()` будет работать и без этого объявления, оно является необязательным. Объявление призвано документировать, что данная функция предназначена для использования внешней переменной.

Функция `local()` показывает, что в случае объявления локальной переменной с тем же именем, что у глобальной, локальная переменная скрывает глобальную переменную. Например, функция `local()` при отображении значения `warming` использует локальное определение переменной `warming`.

Язык C++ расширяет возможности C за счет новой операции разрешения контекста (`::`). Если поместить эту операцию перед именем переменной, будет использоваться глобальная версия этой переменной. Таким образом, `local()` отображает для `warming` значение 0.8, но для `::warming` — значение 0.4. Эта операция еще будет неоднократно встречаться при обсуждении пространств имен и классов. Для обеспечения ясности и во избежание ошибок было бы лучше и безопаснее применять `::warming` в функции `update()` вместо просто `warming`, не полагаясь на правила области видимости.

### Выбор между глобальными и локальными переменными

Теперь, когда имеется возможность выбора между глобальными и локальными переменными, возникает вопрос, каким из них отдать предпочтение? На первый взгляд глобальные переменные кажутся более привлекательными — поскольку все функции имеют к ним доступ, не нужно беспокоиться о передаче аргументов. Однако такой легкий доступ достается дорогой ценой — снижением надежности программ. Опыт показывает, что чем эффективнее программа изолирует данные от нежелательного доступа, тем лучше будет сохраняться их целостность. В большинстве случаев следует пользоваться локальными переменными и передавать данные функциям только по мере необходимости, а не делать данные открытыми за счет использования глобальных переменных. Как вы сможете убедиться позже, объектно-ориентированное программирование делает очередной шаг в плане изоляции данных.

Тем не менее, глобальные переменные имеют свою область применения. Предположим, что имеется блок данных, который должен использоваться несколькими функциями, такой как массив с названиями месяцев или список атомных весов химических элементов. Класс внешнего хранения наилучшим образом подходит для представления константных данных, поскольку в этом случае для предотвращения изменения данных можно воспользоваться ключевым словом `const`:

```
const char * const months[12] =
{
    "January", "February", "March", "April", "May",
    "June", "July", "August", "September", "October",
    "November", "December"
};
```

Первое ключевое слово `const` защищает от изменений строки, а второе слово `const` гарантирует, что каждый указатель в массиве будет постоянно указывать на ту же самую строку, на которую он указывал изначально.

### Статическая продолжительность хранения, внутреннее связывание

Применение модификатора `static` к переменной с областью видимости файла обеспечивает для нее внутреннее связывание. Различие между внутренним и внешним связыванием становится значимым в многофайловых программах. В таком контексте переменная с внутренним связыванием является локальной для файла, который ее содержит. При этом обычная внешняя переменная обладает внешним связыванием, что означает возможность ее применения в различных файлах, как было показано в предыдущем примере.

А что если нужно использовать одно и то же имя для обозначения нескольких переменных в разных файлах? Можно ли просто опустить ключевое слово `extern`?

```
// файл 1
int errors = 20;           // внешнее объявление
...
-----
// файл 2
int errors = 5;           // ??известна только в file2??
void froobish()
{
    cout << errors;       // ошибка
    ...
}
```

Нет, это приведет к ошибке, потому что нарушается правило одного определения. Определение в файле 2 пытается создать внешнюю переменную, так что в программе оказывается два определения `errors`, что является ошибкой.

Однако если в файле объявляется статическая внешняя переменная с тем же именем, что и обычная внешняя переменная, объявленная в другом файле, то в область видимости первого файла попадает статическая версия:

```
// файл 1
int errors = 20;           // внешнее объявление
...
-----
// файл 2
static int errors = 5;     // известна только файлу 2
void froobish()
{
    cout << errors;       // использует переменную errors, определенную в файле 2
    ...
}
```

Это не нарушает правила одного определения, т.к. с помощью ключевого слова `static` для идентификатора `errors` обеспечивается внутреннее связывание, поэтому не предпринимается никаких попыток установить внешнее определение.

#### На заметку!

В многофайловой программе внешнюю переменную можно определять в одном и только одном файле. Все остальные файлы, использующие эту переменную, должны содержать ее объявление с ключевым словом `extern`.

Внешнюю переменную можно применять для разделения данных между различными частями многофайловой программы. Статическую переменную с внутренним связыванием можно использовать для разделения данных между различными функциями в одном файле. (Пространства имен предоставляют для этого альтернативный метод.) Кроме того, если переменную с областью видимости файла сделать `static`, то не придется беспокоиться о конфликте ее имени с переменными с областью видимости файла, которые содержатся в других файлах.

В листингах 9.7 и 9.8 показано, каким образом в C++ поддерживаются переменные с внешним и внутренним связыванием. В листинге 9.7 (`twofile1.cpp`) определены внешние переменные `tom` и `dick`, а также статическая внешняя переменная `harry`. Функция `main()` в этом файле отображает адреса всех трех переменных и затем вызывает функцию `remote_access()`, которая определена во втором файле. Содержимое этого файла (`twofile2.cpp`) приведено в листинге 9.8. Помимо определения функции `remote_access()`, в этом файле с помощью ключевого слова `extern` используется пе-

ременная `tom` из первого файла. Далее в нем определяется статическая переменная по имени `dick`. Модификатор `static` делает эту переменную локальной по отношению к файлу и переопределяет глобальное определение. Затем во втором файле определяется внешняя переменная по имени `harry`. При этом конфликт с переменной `harry` из первого файла не возникает, поскольку она обладает только внутренним связыванием. После этого функция `remote_access()` отображает адреса всех трех переменных, так что их можно сравнить с адресами соответствующих переменных из первого файла. Не забываяте, что для получения готовой программы потребуется скомпилировать и скомпоновать оба файла.

---

### Листинг 9.7. `twofile1.cpp`

```
// twofile1.cpp -- переменные с внешним и внутренним связыванием
#include <iostream>          // должен компилироваться вместе с twofile2.cpp
int tom = 3;                // определение внешней переменной
int dick = 30;             // определение внешней переменной
static int harry = 300;    // статическая, внутреннее связывание

// Прототип функции
void remote_access();

int main()
{
    using namespace std;
    cout << "main() reports the following addresses:\n"; // вывод адресов
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
    remote_access();
    return 0;
}
```

---

### Листинг 9.8. `twofile2.cpp`

```
// twofile2.cpp -- переменные с внутренним и внешним связыванием
#include <iostream>
extern int tom;            // переменная tom определена в другом месте
static int dick = 10;     // переопределяет внешнюю переменную dick
int harry = 200;          // определение внешней переменной,
                          // конфликт с harry из twofile1 отсутствует

void remote_access()
{
    using namespace std;
    cout << "remote_access() reports the following addresses:\n"; // вывод адресов
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
}
```

---

Ниже показан результат выполнения программы из листингов 9.7 и 9.8:

```
main() reports the following addresses:
0x0041a020 = &tom, 0x0041a024 = &dick, 0x0041a028 = &harry
remote_access() reports the following addresses:
0x0041a020 = &tom, 0x0041a450 = &dick, 0x0041a454 = &harry
```

По отображаемым адресам видно, что в обоих файлах используется одна и та же переменная `tom`, но разные переменные `dick` и `harry`. (Значения адресов и формат вывода зависят от системы, в которой выполняется программа. Тем не менее, адреса `tom` будут совпадать друг с другом, тогда как адреса `dick` и `harry` — отличаться.)