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. Обратите внимание, что в проекты добавляются только 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.

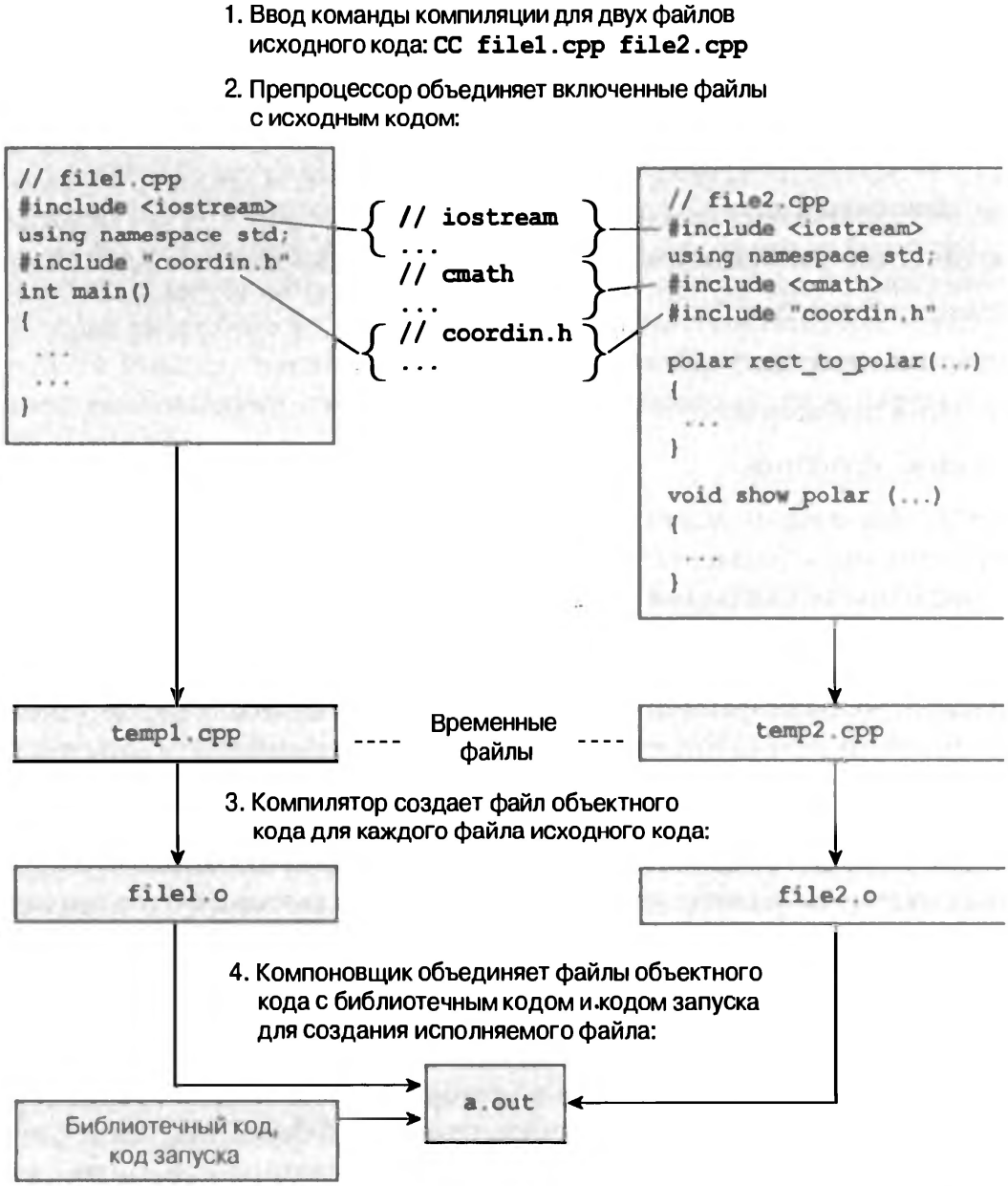

Рис. 9.1. *Компиляция многофайловых программ C++ в системе Unix*

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 xypos);
void show_polar(polar dapos);

#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 внутрь `#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 xypos)
{
   using namespace std;
   polar answer;
   answer.distance =
      sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
   answer.angle = atan2(xypos.y, xypos.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 раздельную 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, сгенерированным 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.