Глава 9

Еще о типах данных и операторах

Прежде чем переходить к более сложным средствам C++, имеет смысл подробнее познакомиться с некоторыми типами данных и операторами. Кроме уже рассмотренных нами типов данных, в C++ определены и другие. Одни из них состоят из модификаторов, добавляемых к уже известным вам типам. Другие включают перечисления, а третьи используют ключевое слово `typedef`. C++ также поддерживает ряд операторов, которые значительно расширяют область действия языка и позволяют решать задачи программирования в весьма широком диапазоне. Речь идет о поразрядных операторах, операторах сдвига, а также операторах "?" и `sizeof`. Кроме того, в этой главе рассматриваются такие специальные операторы, как `new` и `delete`. Они предназначены для поддержки C++-системы динамического распределения памяти.

Спецификаторы типа `const` и `volatile`

> Спецификаторы типа `const` и `volatile` управляют доступом к переменной.

В C++ определено два спецификатора типа, которые оказывают влияние на то, каким образом можно получить доступ к переменным или модифицировать их. Это спецификаторы `const` и `volatile`. Официально они именуются ***cv-спецификаторами*** и должны предшествовать базовому типу при объявлении переменной.

Спецификатор типа `const`

Переменные, объявленные с использованием спецификатора `const`, не могут изменить свои значения во время выполнения программы. Однако любой `const`-переменной можно присвоить некоторое начальное значение. Например, при выполнении инструкции

```
const double version = 3.2;
```

создается `double`-переменная `version`, которая содержит значение 3,2, и это значение программа изменить уже не может. Но эту переменную можно использовать в других выражениях. Любая `const`-переменная получает значение либо во время явно задаваемой инициализации, либо при использовании аппаратно-зависимых средств. Применение спецификатора `const` к объявлению переменной гарантирует, что она не будет модифицирована другими частями вашей программы.

> Спецификатор `const` предотвращает модификацию переменной при выполнении программы.

Спецификатор `const` имеет ряд важных применений. Возможно, чаще всего его используют для создания `const`-параметров типа *указатель*. Такой параметр-указатель защищает объект, на который он ссылается, от модификации со стороны функции. Другими словами, если параметр-указатель предваряется ключевым словом `const`, никакая

инструкция этой функции не может модифицировать переменную, адресуемую этим параметром. Например, функция code() в следующей короткой программе сдвигает каждую букву в сообщении на одну алфавитную позицию (т.е. вместо буквы “А” ставится буква “Б” и т.д.), отображая таким образом сообщение в закодированном виде. Использование спецификатора const в объявлении параметра не позволяет коду функции модифицировать объект, на который указывает этот параметр.

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

void code(const char *str);

int main()
{
   code("Это тест.");
   return 0;
}

/* Использование спецификатора const гарантирует, что str
   не может изменить аргумент, на который он указывает. */
void code(const char *str)
{
   while(*str) {
      cout << (char) (*str+1);
      str++;
   }
}
```

Поскольку параметр str объявляется как const-указатель, у функции code() нет никакой возможности внести изменения в строку, адресуемую параметром str. Но если вы попытаетесь написать функцию code() так, как показано в следующем примере, то обязательно получите сообщение об ошибке, и программа не скомпилируется.

```
// Этот код неверен.
void code(const char *str)
{
   while(*str) {
      *str = *str + 1; // Ошибка, аргумент
                       // модифицировать нельзя.
      cout << (char) *str;
      str++;
   }
}
```

Поскольку параметр str является const-указателем, его нельзя использовать для модификации объекта, на который он ссылается.

Спецификатор const можно также использовать для ссылочных параметров, чтобы не допустить в функции модификацию переменных, на которые ссылаются эти параметры. Например, следующая программа некорректна, поскольку функция f() пытается модифицировать переменную, на которую ссылается параметр i.

```
// Нельзя модифицировать const-ссылки.
#include <iostream>
using namespace std;

void f(const int &i);
```

```
int main()
{
   int k = 10;
   f(k);
   return 0;
}

// Использование ссылочного const-параметра.
void f(const int &i)
{
   i = 100; // Ошибка, нельзя модифицировать const-ссылку.
   cout << i;
}
```

Спецификатор `const` еще можно использовать для подтверждения того, что ваша программа не изменяет значения некоторой переменной. Вспомните, что переменная типа `const` может быть модифицирована внешними устройствами, т.е. ее значение может быть установлено каким-нибудь аппаратным устройством (например, датчиком). Объявив переменную с помощью спецификатора `const`, можно доказать, что любые изменения, которым подвергается эта переменная, вызваны исключительно внешними событиями.

Наконец, спецификатор `const` используется для создания именованных констант. Часто в программах многократно применяется одно и то же значение для различных целей. Например, необходимо объявить несколько различных массивов таким образом, чтобы все они имели одинаковый размер. Когда нужно использовать подобное "магическое число", имеет смысл реализовать его в виде `const`-переменной. Затем вместо реального значения можно использовать имя этой переменной, а если это значение придется впоследствии изменить, вы измените его только в одном месте программы. Следующая программа позволяет попробовать этот вид применения спецификатора `const` "на вкус".

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

const int size = 10;

int main()
{
   int A1[size], A2[size], A3[size];
   // ...
}
```

Если в этом примере понадобится использовать новый размер для массивов, вам потребуется изменить только объявление переменной `size` и перекомпилировать программу. В результате все три массива автоматически получат новый размер.

Спецификатор типа `volatile`

> Спецификатор `volatile` информирует компилятор о том, что данная переменная может быть изменена внешними (по отношению к программе) факторами.

Спецификатор `volatile` сообщает компилятору о том, что значение соответствующей переменной может быть изменено в программе неявным образом. Например, адрес

некоторой глобальной переменной может передаваться управляемой прерываниями подпрограмме тактирования, которая обновляет эту переменную с приходом каждого импульса сигнала времени. В такой ситуации содержимое переменной изменяется без использования явно заданных инструкций программы. Существуют веские основания для того, чтобы сообщить компилятору о внешних факторах изменения переменной. Дело в том, что C++-компилятору разрешается автоматически оптимизировать определенные выражения в предположении, что содержимое той или иной переменной остается неизменным, если оно не находится в левой части инструкции присваивания. Но если некоторые факторы (внешние по отношению к программе) изменят значение этого поля, такое предположение окажется неверным, в результате чего могут возникнуть проблемы.

Например, в следующем фрагменте программы предположим, что переменная `clock` обновляется каждую миллисекунду часовым механизмом компьютера. Но, поскольку переменная `clock` не объявлена с использованием спецификатора `volatile`, этот фрагмент кода может иногда работать недолжным образом. (Обратите особое внимание на строки, обозначенные буквами “А” и “Б”.)

```
int clock, timer;
// ...

timer = clock; // строка А
// ... Какие-нибудь действия.

cout << "Истекшее время " << clock-timer; // строка Б
```

В этом фрагменте переменная `clock` получает свое значение, когда она присваивается переменной `timer` в строке А. Но, поскольку переменная `clock` не объявлена с использованием спецификатора `volatile`, компилятор волен оптимизировать этот код, причем таким способом, при котором значение переменной `clock`, возможно, не будет опрошено в инструкции `cout` (строка Б), если между строками А и Б не будет ни одного промежуточного присваивания значения переменной `clock`. (Другими словами, в строке Б компилятор может просто еще раз использовать значение, которое получила переменная `clock` в строке А.) Но если между моментами выполнения строк А и Б поступят очередные импульсы сигнала времени, то значение переменной `clock` обязательно изменится, а строка Б в этом случае не отразит корректный результат.

Для решения этой проблемы необходимо объявить переменную `clock` с ключевым словом `volatile`.

```
volatile int clock;
```

Теперь значение переменной `clock` будет опрашиваться при каждом ее использовании.

И хотя на первый взгляд это может показаться странным, спецификаторы `const` и `volatile` можно использовать вместе. Например, следующее объявление абсолютно допустимо. Оно создает `const`-указатель на `volatile`-объект.

```
const volatile unsigned char *port = (
                              const volatile char *) 0x2112;
```

В этом примере для преобразования целочисленного литерала `0x2112` в `const`-указатель на `volatile`-символ необходимо применить операцию приведения типов.

Спецификаторы классов памяти

C++ поддерживает пять спецификаторов классов памяти:

```
auto
extern
register
static
mutable
```

> Спецификаторы классов памяти определяют, как должна храниться переменная.

С помощью этих ключевых слов компилятор получает информацию о том, как должна храниться переменная. Спецификатор классов памяти необходимо указывать в начале объявления переменной.

Спецификатор `mutable` применяется только к объектам классов, о которых речь впереди. Остальные спецификаторы мы рассмотрим в этом разделе.

Спецификатор класса памяти `auto`

> Редко используемый спецификатор `auto` объявляет локальную переменную.

Спецификатор `auto` объявляет локальную переменную. Но он используется довольно редко (возможно, вам никогда и не доведется применить его), поскольку локальные переменные являются “автоматическими” по умолчанию. Вряд ли вам попадется это ключевое слово и в чужих программах.

Спецификатор класса памяти `extern`

Все программы, которые мы рассматривали до сих пор, имели довольно скромный размер. Реальные же компьютерные программы гораздо больше. По мере увеличения размера файла, содержащего программу, время компиляции становится иногда раздражающе долгим. В этом случае следует разбить программу на несколько отдельных файлов. После этого небольшие изменения, вносимые в один файл, не потребуют перекомпиляции всей программы. При разработке больших проектов такой многофайловый подход может сэкономить существенное время. Реализовать этот подход позволяет ключевое слово `extern`.

В программах, которые состоят из двух или более файлов, каждый файл должен “знать” имена и типы глобальных переменных, используемых программой в целом. Однако нельзя просто объявить копии глобальных переменных в каждом файле. Дело в том, что в C++ программа может включать только одну копию каждой глобальной переменной. Следовательно, если вы попытаетесь объявить необходимые глобальные переменные в каждом файле, возникнут проблемы. Когда компоновщик попытается скомпоновать эти файлы, он обнаружит дублированные глобальные переменные, и компоновка программы не состоится. Чтобы выйти из этого затруднительного положения, достаточ-

но объявить все глобальные переменные в одном файле, а в других использовать extern-объявления, как показано на рис. 9.1.

Спецификатор extern объявляет переменную, но не выделяет для нее области памяти.

Файл F1	Файл F2

int x, y:
char ch;
 	
int main()
{
  // ...
}
 	
void func1()
{
  x = 123;
}
	extern int x, y:
	extern char ch;
	
	void func22()
	{
	  x = y/10;
	}
	
	void func23()
	{
	  y = 10;
	}

Рис. 9.1. Использование глобальных переменных в отдельно компилируемых модулях

В файле F1 объявляются и определяются переменные x, y и ch. В файле F2 используется скопированный из файла F1 список глобальных переменных, к объявлению которых добавлено ключевое слово extern. Спецификатор extern делает переменную известной для модуля, но в действительности не создает ее. Другими словами, ключевое слово extern предоставляет компилятору информацию о типе и имени глобальных переменных, повторно не выделяя для них памяти. Во время компоновки этих двух модулей все ссылки на эти внешние переменные будут определены.

До сих пор мы не уточняли, в чем состоит различие между объявлением и определением переменной, но здесь это очень важно. При объявлении переменной присваивается имя и тип, а посредством определения для переменной выделяется память. В большинстве случаев объявления переменных одновременно являются определениями. Предварив имя переменной спецификатором extern, можно объявить переменную, не определяя ее.

Существует еще одно применение для ключевого слова extern, которое не связано с многофайловыми проектами. Не секрет, что много времени уходит на объявления глобальных переменных, которые, как правило, приводятся в начале программы, но это не всегда обязательно. Если функция использует глобальную переменную, которая определяется ниже (в том же файле), в теле функции ее можно специфицировать как внешнюю (с помощью ключевого слова extern). При обнаружении определения этой переменной компилятор вычислит соответствующие ссылки на нее.

Рассмотрим следующий пример. Обратите внимание на то, что глобальные переменные first и last объявляются не перед, а после функции main().

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

int main()
{
   extern int first, last; // Использование глобальных
                           // переменных.
```

```
   cout << first << " " << last << "\n";

   return 0;
}

// Глобальное определение переменных first и last.
int first = 10, last = 20;
```

При выполнении этой программы на экран будут выведены числа `10 20`, поскольку глобальные переменные `first` и `last`, используемые в инструкции `cout`, инициализируются этими значениями. Поскольку `extern`-объявление в функции `main()` сообщает компилятору о том, что переменные `first` и `last` объявляются где-то в другом месте (в данном случае ниже, но в том же файле), программу можно скомпилировать без ошибок, несмотря на то, что переменные `first` и `last` используются до их определения.

Важно понимать, что `extern`-объявления переменных, показанные в предыдущей программе, необходимы здесь только по той причине, что переменные `first` и `last` не были определены до их использования в функции `main()`. Если бы их определения компилятор обнаружил раньше определения функции `main()`, необходимости в `extern`-инструкции не было бы. Помните, если компилятор обнаруживает переменную, которая не была объявлена в текущем блоке, он проверяет, не совпадает ли она с какой-нибудь из переменных, объявленных внутри других включающих блоков. Если нет, компилятор просматривает ранее объявленные глобальные переменные. Если обнаруживается совпадение их имен, компилятор предполагает, что ссылка была именно на эту глобальную переменную. Спецификатор `extern` необходим только в том случае, если вы хотите использовать переменную, которая объявляется либо ниже в том же файле, либо в другом.

И еще. Несмотря на то что спецификатор `extern` объявляет, но не определяет переменную, существует одно исключение из этого правила. Если в `extern`-объявлении переменная инициализируется, то такое `extern`-объявление становится определением. Это очень важный момент, поскольку любой объект может иметь несколько объявлений, но только одно определение.

Статические переменные

Переменные типа `static` — это переменные “долговременного” хранения, т.е. они хранят свои значения в пределах своей функции или файла. От глобальных они отличаются тем, что за рамками своей функции или файла они неизвестны. Поскольку спецификатор `static` по-разному определяет “судьбу” локальных и глобальных переменных, мы рассмотрим их в отдельности.

Локальные `static`-переменные

Локальная `static`-переменная поддерживает свое значение между вызовами функции.

Если к локальной переменной применен модификатор `static`, то для нее выделяется постоянная область памяти практически так же, как и для глобальной переменной. Это позволяет статической переменной поддерживать ее значение между вызовами функций.

(Другими словами, в отличие от обычной локальной переменной, значение static-переменной не теряется при выходе из функции.) Ключевое различие между статической локальной и глобальной переменными состоит в том, что статическая локальная переменная известна только блоку, в котором она объявлена. Таким образом, статическую локальную переменную в некоторой степени можно назвать глобальной переменной, которая имеет ограниченную область видимости.

Чтобы объявить статическую переменную, достаточно предварить ее тип ключевым словом static. Например, при выполнении этой инструкции переменная count объявляется статической.

```
static int count;
```

Статической переменной можно присвоить некоторое начальное значение. Например, в этой инструкции переменной count присваивается начальное значение 200:

```
static int count = 200;
```

Локальные static-переменные инициализируются только однажды, в начале выполнения программы, а не при каждом входе в функцию, в которой они объявлены.

Возможность использования статических локальных переменных важна для создания независимых функций, поскольку существуют такие типы функций, которые должны сохранять их значения между вызовами. Если бы статические переменные не были предусмотрены в C++, пришлось бы использовать вместо них глобальные, что открыло бы путь для всевозможных побочных эффектов.

Рассмотрим пример использования static-переменной. Она служит для хранения текущего среднего значения от чисел, вводимых пользователем.

```
/* Вычисляем текущее среднее значение от чисел, вводимых
   пользователем.
*/
#include <iostream>
using namespace std;

int r_avg(int i);

int main()
{
   int num;

   do {
      cout << "Введите числа (-1 означает выход): ";
      cin >> num;
      if(num != -1)
         cout << "Текущее среднее равно: " << r_avg(num);
      cout << '\n';
   } while(num > -1);

   return 0;
}

// Вычисляем текущее среднее.
int r_avg(int i)
{
   static int sum=0, count=0;
   sum = sum + i;
   count++;
   return sum / count;
}
```

Здесь обе локальные переменные `sum` и `count` объявлены статическими и инициализированы значением 0. Помните, что для статических переменных инициализация выполняется только один раз (при первом выполнении функции), а не при каждом входе в функцию. В этой программе функция `r_avg()` используется для вычисления текущего среднего значения от чисел, вводимых пользователем. Поскольку обе переменные `sum` и `count` являются статическими, они поддерживают свои значения между вызовами функции `r_avg()`, что позволяет нам получить правильный результат вычислений. Чтобы убедиться в необходимости модификатора `static`, попробуйте удалить его из программы. После этого программа не будет работать корректно, поскольку промежуточная сумма будет теряться при каждом выходе из функции `r_avg()`.

Глобальные `static`-переменные

> Глобальная `static`-переменная известна только для файла, в котором она объявлена.

Если модификатор `static` применен к глобальной переменной, то компилятор создаст глобальную переменную, которая будет известна только для файла, в котором она объявлена. Это означает, что, хотя эта переменная является глобальной, другие функции в других файлах не имеют о ней “ни малейшего понятия” и не могут изменить ее содержимое. Поэтому она и не может стать “жертвой” несанкционированных изменений. Следовательно, для особых ситуаций, когда локальная статичность оказывается бессильной, можно создать небольшой файл, который будет содержать лишь функции, использующие глобальные `static`-переменные, отдельно скомпилировать этот файл и работать с ним, не опасаясь вреда от побочных эффектов “всеобщей глобальности”.

Рассмотрим пример, который представляет собой переработанную версию программы (из предыдущего раздела), вычисляющей текущее среднее значение. Эта версия состоит из двух файлов и использует глобальные `static`-переменные для хранения значений промежуточной суммы и счетчика вводимых чисел.

```
// --------------------- Первый файл ---------------------
#include <iostream>
using namespace std;

int r_avg(int i);
void reset();

int main()
{
   int num;
   do {
      cout <<
         "Введите числа (-1 для выхода, -2 для сброса): ";
      cin >> num;
      if(num==-2) {
         reset();
         continue;
      }
      if(num != -1)
         cout << "Среднее значение равно: " << r_avg(num);
      cout << '\n';
   } while(num != -1);
```

```
   return 0;
}

// ---------------------- Второй файл ----------------------
static int sum=0, count=0;

int r_avg(int i)
{
   sum = sum + i;
   count++;
   return sum / count;
}

void reset()
{
   sum = 0;
   count = 0;
}
```

В этой версии программы переменные `sum` и `count` являются глобально статическими, т.е. их глобальность ограничена вторым файлом. Итак, они используются функциями `r_avg()` и `reset()`, причем обе они расположены во втором файле. Этот вариант программы позволяет сбрасывать накопленную сумму (путем установки в исходное положение переменных `sum` и `count`), чтобы можно было усреднить другой набор чисел. Но ни одна из функций, расположенных вне второго файла, не может получить доступ к этим переменным. Работая с данной программой, можно обнулить предыдущие накопления, введя число –2. В этом случае будет вызвана функция `reset()`. Проверьте это. Кроме того, попытайтесь получить из первого файла доступ к любой из переменных `sum` или `count`. (Вы получите сообщение об ошибке.)

Итак, имя локальной `static`-переменной известно только функции или блоку кода, в котором она объявлена, а имя глобальной `static`-переменной — только файлу, в котором она "обитает". По сути, модификатор `static` позволяет переменным существовать так, что о них знают только функции, использующие их, тем самым "держа в узде" и ограничивая возможности негативных побочных эффектов. Переменные типа `static` позволяют программисту "скрывать" одни части своей программы от других частей. Это может оказаться просто супердостоинством, когда вам придется разрабатывать очень большую и сложную программу.

Важно! Несмотря на то что глобальные `static`-переменные по-прежнему допустимы и широко используются в C++-коде, стандарт C++ возражает против их применения. Для управления доступом к глобальным переменным рекомендуется другой метод, который заключается в использовании пространств имен. Этот метод описан ниже в этой книге.

Регистровые переменные

Возможно, чаще всего используется спецификатор класса памяти `register`. Для компилятора модификатор `register` означает предписание обеспечить такое хранение соответствующей переменной, чтобы доступ к ней можно было получить максимально

быстро. Обычно переменная в этом случае будет храниться либо в регистре центрального процессора (ЦП), либо в кэше (быстродействующей буферной памяти небольшой емкости). Вероятно, вы знаете, что доступ к регистрам ЦП (или к кэш-памяти) принципиально быстрее, чем доступ к основной памяти компьютера. Таким образом, переменная, сохраняемая в регистре, будет обслужена гораздо быстрее, чем переменная, сохраняемая, например, в оперативной памяти (ОЗУ). Поскольку скорость, с которой к переменным можно получить доступ, определяет, по сути, скорость выполнения вашей программы, для получения удовлетворительных результатов программирования важно разумно использовать спецификатор `register`.

> Спецификатор `register` в объявлении переменной означает требование оптимизировать код для получения максимально возможной скорости доступа к ней.

Формально спецификатор `register` представляет собой лишь запрос, который компилятор вправе проигнорировать. Это легко объяснить: ведь количество регистров (или устройств памяти с малым временем выборки) ограничено, причем для разных сред оно может быть различным. Поэтому, если компилятор исчерпает память быстрого доступа, он будет хранить `register`-переменные обычным способом. В общем случае неудовлетворенный `register`-запрос не приносит вреда, но, конечно же, и не дает никаких преимуществ хранения в регистровой памяти.

Поскольку в действительности только для ограниченного количества переменных можно обеспечить быстрый доступ, важно тщательно выбрать, к каким из них применить модификатор `register`. (Только правильный выбор может повысить быстродействие программы.) Как правило, чем чаще к переменной требуется доступ, тем бо́льшая выгода будет получена в результате оптимизации кода с помощью спецификатора `register`. Поэтому объявлять регистровыми имеет смысл управляющие переменные цикла или переменные, к которым выполняется доступ в теле цикла. На примере следующей функции показано, как `register`-переменная типа `int` используется для управления циклом. Эта функция вычисляет результат выражения m^e для целочисленных значений с сохранением знака исходного числа (т.е. при m = –2 и e = 2 результат будет равен –4).

```
int signed_pwr(register int m, register int e)
{
   register int temp;
   int sign;

   if(m < 0) sign = -1;
   else sign = 1;

   temp = 1;
   for( ;e ;e--) temp = temp * m;

   return temp * sign;
}
```

В этом примере переменные `m`, `e` и `temp` объявлены как регистровые, поскольку все они используются в теле цикла, и потому к ним часто выполняется доступ. Однако переменная `sign` объявлена без спецификатора `register`, поскольку она не является частью цикла и используется реже.

КОПНЕМ ГЛУБЖЕ

Происхождение модификатора `register`

Модификатор `register` был впервые определен в языке C. Первоначально он применялся только к переменным типа `int` и `char` или к указателям и заставлял хранить переменные этого типа в регистре ЦП, а не в ОЗУ, где хранятся обычные переменные. Это означало, что операции с регистровыми переменными могли выполняться намного быстрее, чем операции с остальными (хранимыми в памяти), поскольку для опроса или модификации их значений не требовался доступ к памяти.

После стандартизации языка C было принято решение расширить определение спецификатора `register`. Согласно ANSI-стандарту C модификатор `register` можно применять к любому типу данных. Его использование стало означать для компилятора требование сделать доступ к переменной типа `register` максимально быстрым. Для ситуаций, включающих символы и целочисленные значения, это по-прежнему означает помещение их в регистры ЦП, поэтому традиционное определение все еще в силе. Поскольку язык C++ построен на ANSI-стандарте C, он также поддерживает расширенное определение спецификатора `register`.

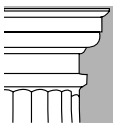
Как упоминалось выше, точное количество `register`-переменных, которые реально будут оптимизированы в любой одной функции, определяется как типом процессора, так и конкретной реализацией C++, которую вы используете. В общем случае можно рассчитывать по крайней мере на две. Однако не стоит беспокоиться о том, что вы могли объявить слишком много `register`-переменных, поскольку C++ автоматически превратит регистровые переменные в нерегистровые, когда их лимит будет исчерпан. (Это гарантирует переносимость C++-кода в рамках широкого диапазона процессоров.)

Чтобы показать влияние, оказываемое `register`-переменными на быстродействие программы, в следующем примере измеряется время выполнения двух циклов `for`, которые отличаются друг от друга только типом управляющих переменных. В программе используется стандартная библиотечная C++-функция `clock()`, которая возвращает количество импульсов сигнала времени системных часов, подсчитанных с начала выполнения этой программы. Программа должна включать заголовок `<ctime>`.

```
/* Эта программа демонстрирует влияние, которое может
   оказать использование register-переменной на скорость
   выполнения программы.
*/

#include <iostream>
#include <ctime>
using namespace std;

unsigned int i; // не register-переменная
unsigned int delay;

int main()
{
   register unsigned int j;
   long start, end;
```

```
   start = clock();
   for(delay=0; delay<50; delay++)
      for(i=0; i < 64000000; i++);
   end = clock();
   cout << "Количество тиков для не register-цикла: ";
   cout << end-start << '\n';

   start = clock();
   for(delay=0; delay<50; delay++)
      for(j=0; j < 64000000; j++) ;
   end = clock();
   cout << "Количество тиков для register-цикла: ";
   cout << end-start << '\n';

   return 0;
}
```

При выполнении этой программы вы убедитесь, что цикл с “регистровым” управлением выполняется приблизительно в два раза быстрее, чем цикл с “нерегистровым” управлением. Если вы не увидели ожидаемой разницы, это может означать, что ваш компилятор оптимизирует все переменные. Просто “поиграйте” программой до тех пор, пока разница не станет очевидной.



На заметку. При написании этой книги была использована среда Visual C++, которая игнорирует ключевое слово `register`. Visual C++ применяет оптимизацию “как считает нужным”. Поэтому вы можете не заметить влияния спецификатора `register` на выполнение предыдущей программы. Однако ключевое слово `register` все еще принимается компилятором без сообщения об ошибке. Оно просто не оказывает никакого воздействия.

Перечисления

В C++ можно определить список именованных целочисленных констант. Такой список называется *перечислением* (enumeration). Эти константы можно затем использовать везде, где допустимы целочисленные значения (например, в целочисленных выражениях). Перечисления определяются с помощью ключевого слова `enum`, а формат их определения имеет такой вид:

```
enum type_name { список_перечисления } список_переменных;
```

Под элементом *список_перечисления* понимается список разделенных запятыми имен, которые представляют значения перечисления. Элемент *список_переменных* необязателен, поскольку переменные можно объявить позже, используя имя типа перечисления. В следующем примере определяется перечисление `apple` и две переменные типа `apple` с именами `red` и `yellow`.

```
enum apple {Jonathan, Golden_Del, Red_Del, Winesap,
            Cortland, McIntosh} red, yellow;
```

Определив перечисление, можно объявить другие переменные этого типа, используя имя перечисления. Например, с помощью следующей инструкции объявляется одна переменная `fruit` перечисления `apple`.

```
apple fruit;
```

Эту инструкцию можно записать и так.

```
enum apple fruit;
```

> Ключевое слово `enum` объявляет перечисление.

Однако использование ключевого слова `enum` здесь излишне. В языке C (который также поддерживает перечисления) обязательной была вторая форма, поэтому в некоторых программах вы можете встретить подобную запись.

С учетом предыдущих объявлений следующие типы инструкций совершенно допустимы.

```
fruit = Winesap;
if(fruit==Red_Del) cout << "Red Delicious\n";
```

Важно понимать, что каждый символ списка перечисления означает целое число, причем каждое следующее число (представленное идентификатором) на единицу больше предыдущего. По умолчанию значение первого символа перечисления равно нулю, следовательно, значение второго — единице и т.д. Поэтому при выполнении этой инструкции

```
cout << Jonathan << ' ' << Cortland;
```

на экран будут выведены числа `0 4`.

Несмотря на то что перечислимые константы автоматически преобразуются в целочисленные, обратное преобразование автоматически не выполняется. Например, следующая инструкция некорректна.

```
fruit = 1; // ошибка
```

Эта инструкция вызовет во время компиляции ошибку, поскольку автоматического преобразования целочисленных значений в значения типа `apple` не существует. Откорректировать предыдущую инструкцию можно с помощью операции приведения типов.

```
fruit = (apple) 1; // Теперь все в порядке,
                   // но стиль не совершенен.
```

Теперь переменная `fruit` будет содержать значение `Golden_Del`, поскольку эта `apple`-константа связывается со значением 1. Как отмечено в комментарии, несмотря на то, что эта инструкция стала корректной, ее стиль оставляет желать лучшего, что простительно лишь в особых обстоятельствах.

Используя инициализатор, можно указать значение одной или нескольких перечислимых констант. Это делается так: после соответствующего элемента списка перечисления ставится знак равенства и нужное целое число. При использовании инициализатора следующему (после инициализированного) элементу списка присваивается значение, на единицу превышающее предыдущее значение инициализатора. Например, при выполнении следующей инструкции константе `Winesap` присваивается значение 10.

```
enum apple {Jonathan, Golden_Del, Red_Del, Winesap=10,
            Cortland, McIntosh};
```

Теперь все символы перечисления `apple` имеют следующие значения.

`Jonathan`	0
`Golden_Del`	1
`Red_Del`	2
`Winesap`	10
`Cortland`	11
`McIntosh`	12

Часто в отношении перечислений ошибочно предполагается, что символы перечисления можно вводить и выводить как строки. Например, следующий фрагмент кода выполнен не будет.

```
// Слово "McIntosh" на экран таким образом не попадет.
fruit = McIntosh;
cout << fruit;
```

Не забывайте, что символ `McIntosh` — это просто имя для некоторого целочисленного значения, а не строка. Следовательно, при выполнении предыдущего кода на экране отобразится числовое значение константы `McIntosh`, а не строка `"McIntosh"`. Конечно, можно создать код ввода и вывода символов перечисления в виде строк, но он выходит несколько громоздким. Вот, например, как можно отобразить на экране названия сортов яблок, связанных с переменной `fruit`.

```
switch(fruit) {
   case Jonathan: cout << "Jonathan";
      break;
   case Golden_Del: cout << "Golden Delicious";
      break;
   case Red_Del: cout << "Red Delicious";
      break;
   case Winesap: cout << "Winesap";
      break;
   case Cortland: cout << "Cortland";
      break;
   case McIntosh: cout << "McIntosh";
      break;
}
```

Иногда для перевода значения перечисления в соответствующую строку можно объявить массив строк и использовать значение перечисления в качестве индекса. Например, следующая программа выводит названия трех сортов яблок.

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

enum apple {Jonathan, Golden_Del, Red_Del, Winesap,
            Cortland, McIntosh};

// Массив строк, связанных с перечислением apple.
char name[][20] = {
   "Jonathan",
   "Golden Delicious",
   "Red Delicious",
   "Winesap",
   "Cortland",
```

```
   "McIntosh"
};

int main()
{
   apple fruit;

   fruit = Jonathan;
   cout << name[fruit] << '\n';

   fruit = Winesap;
   cout << name[fruit] << '\n';

   fruit = McIntosh;
   cout << name[fruit] << '\n';

   return 0;
}
```

Результаты выполнения этой программы таковы.

```
Jonathan
Winesap
McIntosh
```

Использованный в этой программе метод преобразования значения перечисления в строку можно применить к перечислению любого типа, если оно не содержит инициализаторов. Для надлежащего индексирования массива строк перечислимые константы должны начинаться с нуля, быть строго упорядоченными по возрастанию, и каждая следующая константа должна быть больше предыдущей точно на единицу.

Из-за того, что значения перечисления необходимо вручную преобразовывать в удобные для восприятия человеком строки, они, в основном, используются там, где такое преобразование не требуется. Для примера рассмотрите перечисление, используемое для определения таблицы символов компилятора.

Ключевое слово `typedef`

Ключевое слово `typedef` позволяет создать новое имя для существующего типа данных.

В C++ разрешается определять новые имена типов данных с помощью ключевого слова `typedef`. При использовании `typedef`-имени новый тип данных не создается, а лишь определяется новое имя для уже существующего типа. Благодаря `typedef`-именам можно сделать машинозависимые программы более переносимыми: для этого иногда достаточно изменить `typedef`-инструкции. Это средство также позволяет улучшить читабельность кода, поскольку для стандартных типов данных с его помощью можно использовать описательные имена. Общий формат записи инструкции `typedef` таков.

`typedef` *`тип новое_имя`*`;`

Здесь элемент *`тип`* означает любой допустимый тип данных, а элемент *`новое_имя`* — новое имя для этого типа. При этом заметьте: новое имя определяется вами в качестве дополнения к существующему имени типа, а не для его замены.

Например, с помощью следующей инструкции можно создать новое имя для типа `float`.

```
typedef float balance;
```

Эта инструкция является предписанием компилятору распознавать идентификатор `balance` как еще одно имя для типа `float`. После этой инструкции можно создавать `float`-переменные с использованием имени `balance`.

```
balance over_due;
```

Здесь объявлена переменная с плавающей точкой `over_due` типа `balance`, который представляет собой стандартный тип `float`, но имеющий другое название.

Еще об операторах

Выше в этой книге вы уже познакомились с большинством операторов, которые не уникальны для C++. Но, в отличие от других языков программирования, в C++ предусмотрены и другие специальные операторы, которые значительно расширяют возможности языка и повышают его гибкость. Этим операторам и посвящена оставшаяся часть данной главы.

Поразрядные операторы

Поразрядные операторы обрабатывают отдельные биты.

Поскольку C++ нацелен на то, чтобы позволить полный доступ к аппаратным средствам компьютера, важно, чтобы он имел возможность непосредственно воздействовать на отдельные биты в рамках байта или машинного слова. Именно поэтому C++ и содержит поразрядные операторы. Поразрядные операторы предназначены для тестирования, установки или сдвига реальных битов в байтах или словах, которые соответствуют символьным или целочисленным C++-типам. Поразрядные операторы не используются для операндов типа `bool`, `float`, `double`, `long double`, `void` или других еще более сложных типов данных. Поразрядные операторы (они перечислены в табл. 9.1) очень часто используются для решения широкого круга задач программирования системного уровня, например, при опросе информации о состоянии устройства или ее формировании. Теперь рассмотрим каждый оператор этой группы в отдельности.

Таблица 9.1. Поразрядные операторы

Оператор	Значение
&	Поразрядное И (AND)
|	Поразрядное ИЛИ (OR)
^	Поразрядное исключающее ИЛИ (XOR)
>>	Сдвиг вправо
<<	Сдвиг влево
~	Дополнение до 1 (унарный оператор НЕ)

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ (обозначаемые символами `&`, `|`, `^` и `~` соответственно) выполняют те же операции, что и их логические эквиваленты (т.е. они действуют согласно той же таблице истинности). Различие состоит лишь в том, что поразрядные операции работают на побитовой основе. В следующей таблице показан результат выполнения каждой поразрядной операции для всех возможных сочетаний операндов (нулей и единиц).

p	q	p & q	p | q	p ^ q	~p
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Как видно из таблицы, результат применения оператора XOR (исключающее ИЛИ) будет равен значению ИСТИНА (1) только в том случае, если истинен (равен значению 1) лишь один из операндов; в противном случае результат принимает значение ЛОЖЬ (0).

Поразрядный оператор И можно представить как способ подавления битовой информации. Это значит, что 0 в любом операнде обеспечит установку в 0 соответствующего бита результата. Вот пример.

```
    1101 0011
&   1010 1010
    1000 0010
```

Следующая программа считывает символы с клавиатуры и преобразует любой строчный символ в его прописной эквивалент путем установки шестого бита равным значению 0. Набор символов ASCII определен так, что строчные буквы имеют почти такой же код, что и прописные, за исключением того, что код первых отличается от кода вторых ровно на 32. Следовательно, как показано в этой программе, чтобы из строчной буквы сделать прописную, достаточно обнулить ее шестой бит.

```
// Получение прописных букв.
#include <iostream>
using namespace std;

int main()
{
   char ch;

   do {
      cin >> ch;

      // Эта инструкция обнуляет 6-й бит.
      ch = ch & 223; // В переменной ch теперь
                     // прописная буква.

      cout << ch;
```

```
   } while(ch!='Q');

   return 0;
}
```

Значение 223, используемое в инструкции поразрядного И, является десятичным представлением двоичного числа 1101 1111. Следовательно, эта операция И оставляет все биты в переменной `ch` нетронутыми, за исключением шестого (он сбрасывается в нуль).

Оператор И также полезно использовать, если нужно определить, установлен ли интересующий вас бит (т.е. равен ли он значению 1) или нет. Например, при выполнении следующей инструкции вы узнаете, установлен ли 4-й бит в переменной `status`.

```
if(status & 8) cout << "Бит 4 установлен";
```

Чтобы понять, почему для тестирования четвертого бита используется число 8, вспомните, что в двоичной системе счисления число 8 представляется как 0000 1000, т.е. в числе 8 установлен только четвертый разряд. Поэтому условное выражение инструкции `if` даст значение ИСТИНА только в том случае, если четвертый бит переменной `status` также установлен (равен 1). Интересное использование этого метода показано на примере функции `disp_binary()`. Она отображает в двоичном формате конфигурацию битов своего аргумента. Мы будем использовать функцию `disp_binary()` ниже в этой главе для исследования возможностей других поразрядных операций.

```
// Отображение конфигурации битов в байте.
void disp_binary(unsigned u)
{
   register int t;

   for(t=128; t>0; t = t/2)
      if(u & t) cout << "1 ";
      else cout << "0 ";
   cout << "\n";
}
```

Функция `disp_binary()`, используя поразрядный оператор И, последовательно тестирует каждый бит младшего байта переменной `u`, чтобы определить, установлен он или сброшен. Если он установлен, отображается цифра 1, в противном случае — цифра 0. Интереса ради попробуйте расширить эту функцию так, чтобы она отображала все биты переменной `u`, а не только ее младший байт.

Поразрядный оператор ИЛИ, в противоположность поразрядному И, удобно использовать для установки нужных битов в единицу. При выполнении операции ИЛИ наличие в любом операнде бита, равного 1, означает, что в результате соответствующий бит также будет равен единице. Вот пример.

```
  1101 0011
| 1010 1010
  1111 1011
```

Можно использовать оператор ИЛИ для превращения рассмотренной выше программы (которая преобразует строчные символы в их прописные эквиваленты) в ее “противоположность”, т.е. теперь, как показано ниже, она будет преобразовывать прописные буквы в строчные.

```
// Получение строчных букв.
#include <iostream>
```

```
using namespace std;

int main()
{
   char ch;

   do {
      cin >> ch;

      /* Эта инструкция делает букву строчной,
         устанавливая ее 6-й бит.
      */
      ch = ch | 32;

      cout << ch;
   } while(ch != 'q');

   return 0;
}
```

Установка шестого бита превращает прописную букву в ее строчный эквивалент.

Поразрядное исключающее ИЛИ (XOR) устанавливает в единицу бит результата только в том случае, если соответствующие биты операндов отличаются один от другого, т.е. не равны. Вот пример:

```
  0111 1111
^ 1011 1001
  1100 0110
```

Унарный оператор НЕ (или оператор дополнения до 1) инвертирует состояние всех битов своего операнда. Например, если целочисленное значение (хранимое в переменной `A`), представляет собой двоичный код 1001 0110, то в результате операции `~A` получим двоичный код 0110 1001.

В следующей программе демонстрируется использование оператора НЕ посредством отображения некоторого числа и его дополнения до 1 в двоичном коде с помощью приведенной выше функции `disp_binary()`.

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

void disp_binary(unsigned u);

int main()
{
   unsigned u;

   cout << "Введите число между 0 и 255: ";
   cin >> u;

   cout << "Исходное число в двоичном коде: ";
   disp_binary(u);

   cout << "Его дополнение до единицы: ";
   disp_binary(~u);

   return 0;
}
```

```
// Отображение битов, составляющих байт.
void disp_binary(unsigned u)
{
   register int t;

   for(t=128; t>0; t = t/2)
      if(u & t) cout << "1 ";
      else cout << "0 ";
   cout << "\n";
}
```

Вот как выглядят результаты выполнения этой программы.

```
Введите число между 0 и 255: 99
Исходное число в двоичном коде: 0 1 1 0 0 0 1 1
Его дополнение до единицы: 1 0 0 1 1 1 0 0
```

И еще. Не путайте логические и поразрядные операторы. Они выполняют различные действия. Операторы `&`, `|` и `~` применяются непосредственно к каждому биту значения в отдельности. Эквивалентные логические операторы обрабатывают в качестве операндов значения ИСТИНА/ЛОЖЬ (не нуль/нуль). Поэтому поразрядные операторы нельзя использовать вместо их логических эквивалентов в условных выражениях. Например, если значение `x` равно 7, то выражение `x && 8` имеет значение ИСТИНА, в то время как выражение `x & 8` дает значение ЛОЖЬ.

Узелок на память. Оператор отношения или логический оператор всегда генерирует результат, который имеет значение ИСТИНА или ЛОЖЬ, в то время как аналогичный поразрядный оператор генерирует значение, получаемое согласно таблице истинности конкретной операции.

Операторы сдвига

Операторы сдвига, “>>” и “<<”, сдвигают все биты в значении вправо или влево. Общий формат использования оператора сдвига вправо выглядит так.

значение `>>` *число_битов*

А оператор сдвига влево используется так.

значение `<<` *число_битов*

Операторы сдвига предназначены для сдвига битов в рамках целочисленного значения.

Здесь элемент *число_битов* указывает, на сколько позиций должно быть сдвинуто *значение*. При каждом сдвиге влево все биты, составляющее значение, сдвигаются влево на одну позицию, а в младший разряд записывается нуль. При каждом сдвиге вправо все биты сдвигаются, соответственно, вправо. Если сдвигу вправо подвергается значение без знака, в старший разряд записывается нуль. Если же сдвигу вправо подвергается значение со знаком, значение знакового разряда сохраняется. Как вы помните, отрицательные целые числа представляются установкой старшего разряда числа равным единице. Таким образом, если сдвигаемое значение отрицательно, при каждом сдвиге вправо в старший разряд записывается единица, а если положительно — нуль. Не забывайте, сдвиг, выполняемый операторa-

ми сдвига, не является циклическим, т.е. при сдвиге как вправо, так и влево крайние биты теряются, и содержимое потерянного бита узнать невозможно.

Операторы сдвига работают только со значениями целочисленных типов, например, символами, целыми числами и длинными целыми числами. Они не применимы к значениям с плавающей точкой.

Побитовые операции сдвига могут оказаться весьма полезными для декодирования входной информации, получаемой от внешних устройств (например, цифроаналоговых преобразователей), и обработки информация о состоянии устройств. Поразрядные операторы сдвига можно также использовать для выполнения ускоренных операций умножения и деления целых чисел. С помощью сдвига влево можно эффективно умножать на два, сдвиг вправо позволяет не менее эффективно делить на два.

Следующая программа наглядно иллюстрирует результат использования операторов сдвига.

```
// Демонстрация выполнения поразрядного сдвига.
#include <iostream>
using namespace std;

void disp_binary(unsigned u);
int main()
{
   int i=1, t;
   for(t=0; t<8; t++) {
      disp_binary(i);
      i = i << 1;
   }

   cout << "\n";

   for(t=0; t<8; t++) {
      i = i >> 1;
      disp_binary(i);
   }

   return 0;
}

// Отображение битов, составляющих байт.
void disp_binary(unsigned u)
{
   register int t;

   for(t=128; t>0; t=t/2)
      if(u & t) cout << "1 ";
      else cout << "0 ";
   cout << "\n";
}
```

Результаты выполнения этой программы таковы.

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
```

```
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

Оператор “знак вопроса”

Одним из самых замечательных операторов C++ является оператор “?”. Оператор “?” можно использовать в качестве замены `if-else`-инструкций, употребляемых в следующем общем формате.

```
if (условие)
   переменная = выражение1;
else
   переменная = выражение2;
```

Здесь значение, присваиваемое переменной, зависит от результата вычисления элемента *условие*, управляющего инструкцией `if`.

Оператор “?” называется *тернарным*, поскольку он работает с тремя операндами. Вот его общий формат записи:

```
Выражение1 ? Выражение2 : Выражение3;
```

Все элементы здесь являются выражениями. Обратите внимание на использование и расположение двоеточия.

Значение ?-выражения определяется следующим образом. Вычисляется *Выражение1*. Если оно оказывается истинным, вычисляется *Выражение2*, и результат его вычисления становится значением всего ?-выражения. Если результат вычисления элемента *Выражение1* оказывается ложным, значением всего ?-выражения становится результат вычисления элемента *Выражение3*. Рассмотрим следующий пример.

```
while(something) {
   x = count > 0 ? 0 : 1;
   // ...
}
```

Здесь переменной `x` будет присваиваться значение 0 до тех пор, пока значение переменной `count` не станет меньше или равно нулю. Аналогичный код (но с использованием `if-else`-инструкции) выглядел бы так.

```
while(something) {
   if(count > 0) x = 0;
   else x = 1;
   // ...
}
```

А вот еще один пример практического применения оператора `?`. Следующая программа делит два числа, но не допускает деления на нуль.

```
/* Эта программа использует оператор ? для предотвращения
   деления на нуль. */
```

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

int div_zero();

int main()
{
   int i, j, result;

   cout << "Введите делимое и делитель: ";
   cin >> i >> j;

   // Эта инструкция не допустит возникновения ошибки
   // деления на нуль.

   result = j ? i/j : div_zero();

   cout << "Результат: " << result;

   return 0;
}

int div_zero()
{
   cout << "Нельзя делить на нуль.\n";
   return 0;
}
```

Здесь, если значение переменной `j` не равно нулю, выполняется деление значения переменной `i` на значение переменной `j`, а результат присваивается переменной `result`. В противном случае вызывается обработчик ошибки деления на нуль `div_zero()`, и переменной `result` присваивается нулевое значение.

Составные операторы присваивания

В C++ предусмотрены специальные составные операторы присваивания, в которых объединено присваивание с еще одной операцией. Начнем с примера и рассмотрим следующую инструкцию.

```
x = x + 10;
```

Используя составной оператор присваивания, ее можно переписать в таком виде.

```
x += 10;
```

Пара операторов `+=` служит указанием компилятору присвоить переменной `x` сумму текущего значения переменной `x` и числа 10. Этот пример служит иллюстрацией того, что составные операторы присваивания упрощают программирование определенных инструкций присваивания. Кроме того, они позволяют компилятору сгенерировать более эффективный код.

Составные версии операторов присваивания существуют для всех бинарных операторов (т.е. для всех операторов, которые работают с двумя операндами). Таким образом, при таком общем формате бинарных операторов присваивания

```
переменная = переменная op выражение;
```

общая форма записи их составных версий выглядит так:

```
переменная op = выражение;
```

Здесь элемент *op* означает конкретный арифметический или логический оператор, объединяемый с оператором присваивания.

А вот еще один пример. Инструкция

```
x = x - 100;
```

аналогична такой:

```
x -= 100;
```

Обе эти инструкции присваивают переменной `x` ее прежнее значение, уменьшенное на 100.

Составные операторы присваивания можно часто встретить в профессионально написанных C++-программах, поэтому каждый C++-программист должен быть с ними на “ты” .

Оператор “запятая”

Не менее интересным, чем описанные выше операторы, является такой оператор C++, как “запятая”. Вы уже видели несколько примеров его использования в цикле `for`, где с его помощью была организована инициализация сразу нескольких переменных. Но оператор “запятая” также может составлять часть выражения. Его назначение в этом случае — связать определенным образом несколько выражений. Значение списка выражений, разделенных запятыми, определяется в этом случае значением крайнего справа выражения. Значения других выражений отбрасываются. Следовательно, значение выражения справа становится значением всего выражения-списка. Например, при выполнении этой инструкции

```
var = (count=19, incr=10, count+1);
```

переменной `count` сначала присваивается число 19, переменной `incr` — число 10, а затем к значению переменной `count` прибавляется единица, после чего переменной `var` присваивается значение крайнего справа выражения, т.е. `count+1`, которое равно 20. Круглые скобки здесь обязательны, поскольку оператор “запятая” имеет более низкий приоритет, чем оператор присваивания.

Чтобы понять назначение оператора “запятая”, попробуем выполнить следующую программу.

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

int main()
{
   int i, j;
   j = 10;
   i = (j++, j+100, 999+j);
   cout << i;
   return 0;
}
```

Эта программа выводит на экран число 1010. И вот почему: сначала переменной `j` присваивается число 10, затем переменная `j` инкрементируется до 11. После этого вычисляется выражение `j+100`, которое нигде не применяется. Наконец, выполняется сложение значения переменной `j` (оно по-прежнему равно 11) с числом 999, что в результате дает число 1010.

По сути, назначение оператора "запятая" — обеспечить выполнение заданной последовательности операций. Если эта последовательность используется в правой части инструкции присваивания, то переменной, указанной в ее левой части, присваивается значение последнего выражения из списка выражений, разделенных запятыми. Оператор "запятая" по его функциональной нагрузке можно сравнить со словом "и", используемым в фразе: "сделай это, *и* то, *и* другое...".

Несколько присваиваний "в одном"

Язык C++ позволяет применить очень удобный метод одновременного присваивания многим переменным одного и того же значения. Речь идет об объединении сразу нескольких присваиваний в одной инструкции. Например, при выполнении этой инструкции переменным `count`, `incr` и `index` будет присвоено число 10.

```
count = incr = index = 10;
```

Этот формат присвоения нескольким переменным общего значения можно часто встретить в профессионально написанных программах.

Использование ключевого слова `sizeof`

Иногда полезно знать размер (в байтах) одного из типов данных. Поскольку размеры встроенных C++-типов данных в разных вычислительных средах могут быть различными, а знание размера переменной во всех ситуациях имеет важное значение, то для решения этой проблемы в C++ включен оператор (действующий во время компиляции программы), который используется в двух следующих форматах.

```
sizeof (type)
sizeof value
```

> Оператор `sizeof` во время компиляции программы получает размер типа или значения.

Первая версия возвращает размер заданного типа данных, а вторая — размер заданного значения. Если вам нужно узнать размер некоторого типа данных (например, `int`), заключите название этого типа в круглые скобки. Если же вас интересует размер области памяти, занимаемой конкретным значением, можно обойтись без круглых скобок, хотя при желании их можно использовать.

Чтобы понять, как работает оператор `sizeof`, испытайте следующую короткую программу. Для многих 32-разрядных сред она должна отобразить значения 1, 4, 4 и 8.

```
// Демонстрация использования оператора sizeof.
#include <iostream>
using namespace std;

int main()
{
   char ch;
   int i;

   cout << sizeof ch << ' ';       // размер типа char
   cout << sizeof i << ' ';        // размер типа int
   cout << sizeof (float) << ' ';  // размер типа float
   cout << sizeof (double) << ' '; // размер типа double

   return 0;
}
```

Как упоминалось выше, оператор `sizeof` действует во время компиляции программы. Вся информация, необходимая для вычисления размера указанной переменной или заданного типа данных, известна уже во время компиляции.

Оператор `sizeof` можно применить к любому типу данных. Например, в случае применения к массиву он возвращает количество байтов, занимаемых массивом. Рассмотрим следующий фрагмент кода.

```
int nums[4];
cout << sizeof nums; // Будет выведено число 16.
```

Для 4-байтных значений типа `int` при выполнении этого фрагмента кода на экране отобразится число 16 (которое получается в результате умножения 4 байт на 4 элемента массива).

Оператор `sizeof` главным образом используется при написании кода, который зависит от размера C++-типов данных. Помните: поскольку размеры типов данных в C++ определяются конкретной реализацией, не стоит полагаться на размеры типов, определенные в реализации, в которой вы работаете в данный момент.

Динамическое распределение памяти с использованием операторов `new` и `delete`

Для C++-программы существует два основных способа хранения информации в основной памяти компьютера. Первый состоит в использовании переменных. Область памяти, предоставляемая переменным, закрепляется за ними во время компиляции и не может быть изменена при выполнении программы. Второй способ заключается в использовании C++-системы динамического распределения памяти. В этом случае память для данных выделяется по мере необходимости из раздела свободной памяти, который расположен между вашей программой (и ее постоянной областью хранения) и стеком. Этот

раздел называется *"кучей"* (heap). (Расположение программы в памяти схематично показано на рис. 9.2.)

> Система динамического распределения памяти — это средство получения программой некоторой области памяти во время ее выполнения.

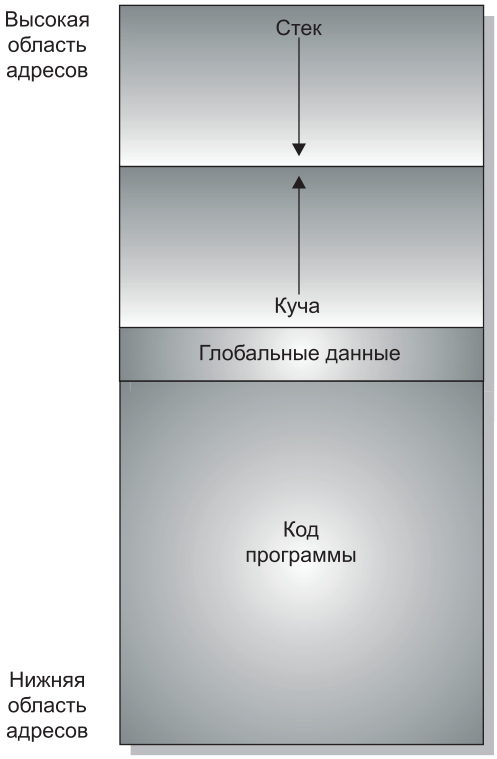


Рис. 9.2. Использование памяти в C++-программе

Динамическое выделение памяти — это получение программой памяти во время ее выполнения. Другими словами, благодаря этой системе программа может создавать переменные во время выполнения, причем в нужном (в зависимости от ситуации) количестве. Эта система динамического распределения памяти особенно ценна для таких структур данных, как связные списки и двоичные деревья, которые изменяют свой размер по мере их использования. Динамическое выделение памяти для тех или иных целей — важная составляющая почти всех реальных программ.

Чтобы удовлетворить запрос на динамическое выделение памяти, используется так называемая "куча". Нетрудно предположить, что в некоторых чрезвычайных ситуациях свободная память "кучи" может исчерпаться. Следовательно, несмотря на то, что динамическое распределение памяти (по сравнению с фиксированным) обеспечивает бо́льшую гибкость, но и в этом случае оно имеет свои пределы.

Оператор `new` позволяет динамически выделить область памяти.

Язык C++ содержит два оператора, `new` и `delete`, которые выполняют функции по выделению и освобождению памяти. Приводим их общий формат.

```
переменная-указатель = new тип_переменной;
delete переменная-указатель;
```

Оператор `delete` освобождает ранее выделенную динамическую память.

Здесь элемент *переменная-указатель* представляет собой указатель на значение, тип которого задан элементом *тип_переменной*. Оператор `new` выделяет область памяти, достаточную для хранения значения заданного типа, и возвращает указатель на эту область памяти. С помощью оператора `new` можно выделить память для значений любого допустимого типа. Оператор `delete` освобождает область памяти, адресуемую заданным указателем. После освобождения эта память может быть снова выделена в других целях при последующем `new`-запросе на выделение памяти.

Поскольку объем "кучи" конечен, она может когда-нибудь исчерпаться. Если для удовлетворения очередного запроса на выделение памяти не существует достаточно свободной памяти, оператор `new` потерпит фиаско, и будет сгенерировано исключение. *Исключение* — это ошибка специального типа, которая возникает во время выполнения программы (в C++ предусмотрена целая подсистема, предназначенная для обработки таких ошибок). (Исключения описаны в главе 17.) В общем случае ваша программа должна обработать подобное исключение и по возможности выполнить действие, соответствующее конкретной ситуации. Если это исключение не будет обработано вашей программой, ее выполнение будет прекращено.

Такое поведение оператора `new` в случае невозможности удовлетворить запрос на выделение памяти определено стандартом C++. На такую реализацию настроены также все современные компиляторы, включая последние версии Visual C++ и C++ Builder. Однако дело в том, что некоторые более ранние компиляторы обрабатывают `new`-инструкции по-другому. Сразу после изобретения языка C++ оператор `new` при неудачном выполнении возвращал нулевой указатель. Позже его реализация была изменена так, чтобы в случае неудачи генерировалось исключение, как было описано выше. Поскольку в этой книге мы придерживаемся стандарта C++, то во всех представленных здесь примерах предполагается именно генерирование исключения. Если же вы используете более старый компилятор, обратитесь к прилагаемой к нему документации и уточните, как реализован оператор `new` (при необходимости внесите в примеры соответствующие изменения).

Поскольку исключения рассматриваются ниже в этой книге (после темы классов и объектов), мы не будем пока отвлекаться на обработку исключений, генерируемых в случае неудачного выполнения оператора `new`. Кроме того, ни один из примеров в этой и последующих главах не должен вызвать неудачного выполнения оператора `new`, поскольку в этих программах запрашивается лишь несколько байтов. Но если такая ситуация все же возникнет, то в худшем случае это приведет к завершению программы. В главе 17, посвященной обработке исключений, вы узнаете, как обработать исключение, сгенерированное оператором `new`.

Рассмотрим пример программы, которая иллюстрирует использование операторов new и delete.

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

int main()
{
   int *p;

   p = new int; // Выделяем память для int-значения.

   *p = 20;     // Помещаем в эту область памяти
                // значение 20.
   cout << *p;  // Убеждаемся (путем вывода на экран) в
                // работоспособности этого кода.
   delete p;    // Освобождаем память.

   return 0;
}
```

Эта программа присваивает указателю p адрес (взятой из "кучи") области памяти, которая будет иметь размер, достаточный для хранения целочисленного значения. Затем в эту область памяти помещается число 20, после чего на экране отображается ее содержимое. Наконец, динамически выделенная память освобождается.

Благодаря такому способу организации динамического выделения памяти оператор delete необходимо использовать только с тем указателем на память, который был возвращен в результате new-запроса на выделение памяти. Использование оператора delete с другим типом адреса может вызвать серьезные проблемы.

Инициализация динамически выделенной памяти

Используя оператор new, динамически выделяемую память можно инициализировать. Для этого после имени типа задайте начальное значение, заключив его в круглые скобки. Например, в следующей программе область памяти, адресуемая указателем p, инициализируется значением 99.

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

int main()
{
   int *p;

   p = new int (99); // Инициализируем память числом 99.
   cout << *p;       // На экран выводится число 99.

   delete p;
   return 0;
}
```

Выделение памяти для массивов

С помощью оператора new можно выделять память и для массивов. Вот как выглядит общий формат операции выделения памяти для одномерного массива:

переменная-указатель = new *тип* [*размер*];

Здесь элемент *размер* задает количество элементов в массиве.

Чтобы освободить память, выделенную для динамически созданного массива, используйте такой формат оператора delete:

delete [] *переменная-указатель*;

Здесь элемент *переменная-указатель* представляет собой адрес, полученный при выделении памяти для массива (с помощью оператора new). Квадратные скобки означают для C++, что динамически созданный массив удаляется, а вся область памяти, выделенная для него, автоматически освобождается.

Важно! Более старые C++-компиляторы могут требовать задания размера удаляемого массива, поскольку в ранних версиях C++ для освобождения памяти, занимаемой удаляемым массивом, необходимо было применять такой формат оператора delete:

delete [*размер*] *переменная-указатель*;

Здесь элемент *размер* задает количество элементов в массиве. Стандарт C++ больше не требует указывать размер при его удалении.

При выполнении следующей программы выделяется память для 10-элементного массива типа double, который затем заполняется значениями от 100 до 109, после чего содержимое этого массива отображается на экране.

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

int main()
{
   double *p;
   int i;

   p = new double [10]; // Выделяем память для
                        // 10-элементного массива.

   // Заполняем массив значениями от 100 до 109.
   for(i=0; i<10; i++) p[i] = 100.00 + i;

   // Отображаем содержимое массива.
   for(i=0; i<10; i++) cout << p[i] << " ";

   delete [] p; // Удаляем весь массив.

   return 0;
}
```

При динамическом выделении памяти для массива важно помнить, что его нельзя одновременно и инициализировать.

КОПНЕМ ГЛУБЖЕ

Динамическое распределение памяти в языке C: функции `malloc()` и `free()`

Язык C не содержит операторов `new` или `delete`. Вместо них в C используются библиотечные функции, предназначенные для выделения и освобождения памяти. В целях совместимости C++ по-прежнему поддерживает C-систему динамического распределения памяти и не зря: в C++-программах все еще используются C-ориентированные средства динамического распределения памяти. Поэтому им стоит уделить внимание.

Ядро C-системы распределения памяти составляют функции `malloc()` и `free()`. Функция `malloc()` предназначена для выделения памяти, а функция `free()` — для ее освобождения. Другими словами, каждый раз, когда с помощью функции `malloc()` делается запрос, часть свободной памяти выделяется в соответствии с этим запросом. При каждом вызове функции `free()` соответствующая область памяти возвращается системе. Любая программа, которая использует эти функции, должна включать заголовок `<cstdlib>`.

Функция `malloc()` имеет такой прототип.

```
void *malloc(size_t num_bytes);
```

Здесь *`num_bytes`* означает количество байтов запрашиваемой памяти. (Тип `size_t` представляет собой разновидность целочисленного типа без знака.) Функция `malloc()` возвращает указатель типа `void`, который играет роль обобщенного указателя. Чтобы из этого обобщенного указателя получить указатель на нужный вам тип, необходимо использовать операцию приведения типов. В результате успешного вызова функция `malloc()` возвратит указатель на первый байт области памяти, выделенной из "кучи". Если для удовлетворения запроса свободной памяти в системе недостаточно, функция `malloc()` возвращает нулевой указатель.

Функция `free()` выполняет действие, обратное действию функции `malloc()` в том, что она возвращает системе ранее выделенную ею память. После освобождения память можно снова использовать последующим обращением к функции `malloc()`. Функция `free()` имеет такой прототип.

```
void free(void *ptr);
```

Здесь параметр *`ptr`* представляет собой указатель на память, ранее выделенную с помощью функции `malloc()`. Никогда не следует вызывать функцию `free()` с недействительным аргументом; это может привести к разрушению списка областей памяти, подлежащих освобождению.

Использование функций `malloc()` и `free()` иллюстрируется в следующей программе.

```
// Демонстрация использования функций malloc() и free().
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
   int *i;
```

```
   double *j;

   i = (int *) malloc(sizeof(int));
   if(!i) {
      cout << "Выделить память не удалось.\n";
      return 1;
   }

   j = (double *) malloc(sizeof(double));
   if(!j) {
      cout << "Выделить память не удалось.\n";
      return 1;
   }

   *i= 10;
   *j = 100.123;

   cout << *i << ' ' << *j;

   // Освобождение памяти.
   free(i);
   free(j);

   return 0;
}
```

Несмотря на то что функции `malloc()` и `free()` — полностью пригодны для динамического распределения памяти, есть ряд причин, по которым в C++ определены собственные средства динамического распределения памяти. Во-первых, оператор `new` автоматически вычисляет размер выделяемой области памяти для заданного типа, т.е. вам не нужно использовать оператор `sizeof`, а значит, налицо экономия в коде и трудовых затратах программиста. Но важнее то, что автоматическое вычисление не допускает выделения неправильного объема памяти. Во-вторых, C++-оператор `new` автоматически возвращает корректный тип указателя, что освобождает программиста от необходимости использовать операцию приведения типов. В-третьих, используя оператор `new`, можно инициализировать объект, для которого выделяется память. Наконец, как будет показано ниже в этой книге, программист может создать собственные версии операторов `new` и `delete`.

И последнее. Из-за возможной несовместимости не следует смешивать функции `malloc()` и `free()` с операторами `new` и `delete` в одной программе.

Сводная таблица приоритетов C++-операторов

В табл. 9.2 показан приоритет выполнения всех C++-операторов (от высшего до самого низкого). Большинство операторов ассоциированы слева направо. Но унарные операторы, операторы присваивания и оператор "?" ассоциированы справа налево. Обратите внимание на то, что эта таблица включает несколько операторов, которые мы пока не использовали в наших примерах, поскольку они относятся к объектно-ориентированному программированию (и описаны ниже).

Таблица 9.2. Приоритет C++-операторов

Наивысший	`() [] -> :: .`
	`! ~ ++ -- - * & sizeof new delete typeid` операторы приведения типа
	`.* ->*`
	`* / %`
	`+ -`
	`<< >>`
	`< <= > >=`
	`== !=`
	`&`
	`^`
	`|`
	`&&`
	`||`
	`?:`
	`= += -= *= /= %= >>= <<= &= ^= |=`
Низший	`,`