Глава 4
Массивы, указатели и ссылки

В этой главе исследуется несколько важных аспектов применения массивов объектов и указателей на объекты. Заканчивается глава обсуждением одного из самых важных нововведений C++ - ссылок. Ссылки определяют многие возможности C++, поэтому при чтении нужно быть особенно внимательным.

4.1. Массивы объектов

Как уже отмечалось ранее, объекты — это переменные, и они имеют те же возможности и признаки, что и переменные любых других типов. Поэтому вполне допустимо упаковывать объекты в массив. Синтаксис объявления массива объектов совершенно аналогичен тому, который используется для объявления массива переменных любого другого типа. Более того, доступ к массивам объектов совершенно аналогичен доступу к массивам переменных любого другого типа.

Примеры

1. Пример массива объектов:

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

class samp {
   int a;
public:
   void set_a(int n) { a = n; }
   int get_a() { return a; }
};

int main()
{
   samp ob[4];
   int i;

   for(i=0; i<4; i++) ob[ i ].set_a(i);

   for(i=0; i<4; i++) cout << ob[ i ].get_a();

   cout << "\n";

   return 0;
}
```

В этой программе создается массив из четырех элементов типа **samp,** которым затем присваиваются значения от 0 до 3. Обратите внимание на то, как вызываются функции-члены для каждого элемента массива. Имя массива, в данном случае **ob,** индексируется; затем применяется оператор доступа к члену, за которым следует имя вызываемой функции-члена.

2. Если класс содержит конструктор, массив объектов может быть инициализирован. Например, здесь объект **ob** является инициализируемым массивом:

```
// Инициализация массива
#include <iostream>
using namespace std;

class samp {
   int a;
public:
   samp(int n) { a = n; }
   int get_a() { return a; }
};

int main()
{
   samp ob[4] = { -1, -2, -3, -4 };
   int i;

   for(i=0; i<4; i++) cout << ob[ i ].get_a() << ' ';

   cout << "\n";

   return 0;
}
```

Эта программа выводит на экран —1 —2 —3 —4. В этом примере-значения —1 до—4 передаются объекту **ob** конструктором.

Фактически синтаксис списка инициализации — это сокращение следующей конструкции (впервые показанной в главе 2):

```
samp ob[4] = { samp(-l), samp(-2), samp(-3), samp(-4) };
```

Однако при инициализации одномерного массива общепринятой является та форма записи, которая была показана в программе (хотя, как вы дальше увидите, такая форма записи будет работать только с теми массивами, конструктор которых имеет единственный аргумент).

3. Вы также можете работать с многомерными массивами объектов. Например, эта программа создает двумерный массив объектов и инициализирует его:

```
// Создание двумерного массива объектов
#include <iostream>
using namespace std;

class samp {
   int a;
public:
   samp(int n) { a = n; }
   int get_a() { return a; }
};

int main()
{
   samp ob[4] [2] = {
      1, 2,
      3, 4,
      5, 6,
      7, 8
    };
   int i;
```

```
  for(i=0; i<4; i++) {
    cout << ob[i] [0].get_a() << ' ';
    cout << ob[i] [1].get_a() << "\n";
  }

  cout << "\n";

  return 0;
}
```

Эта программа выводит на экран следующее:

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

4. Как вы знаете, конструктор может иметь более одного аргумента. При инициализации массива объектов с таким конструктором вы должны использовать упоминавшуюся ранее альтернативную форму инициализации. Начнём с примера:

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

class samp {
  int a, b;
public:
  samp(int n, int m) { a = n; b = m; }
  int get_a() { return a; }
  int get_b() { return b; }
};

int main()
{
  samp ob[4] [2] = {
    samp(1, 2), samp(3, 4),
    samp(5, 6), samp(7, 8),
    samp(9, 10), samp(11, 12),
    samp(13, 14), samp(15, 16)
   };
  int i;

  for(i=0; i<4; i++) {
    cout << ob[i] [0].get_a() << ' ';
    cout << ob[i] [0].get_b() << "\n";
    cout << ob[i] [1].get_a() << ' ';
    cout << ob[i] [1].get_b() << "\n";
  }

  cout << "\n";

  return 0;
}
```

В этом примере конструктор **samp** имеет два аргумента. Здесь массив **ob** объявляется и инициализируется в функции **main()** с помощью прямых вызовов конструктора **samp.** Это необходимо, поскольку формальный синтаксис C++ позволяет одновременное использование только одного аргумента в разделяемом запятыми списке. При этом невозможно задать, например, два или более аргумента в каждом элементе списка. Поэтому если вы инициализируете массив объектов, имеющих конструктор с более чем

одним аргументом, то вам следует пользоваться длинной формой инициализации вместо ее сокращенной формы.

14.2. Использование указателей на объекты

Как отмечалось в главе 2, доступ к объекту можно получить через указатель на этот объект. Как вы знаете, при использовании указателя на объект к членам объекта обращаются не с помощью оператора точка (.), а с помощью оператора стрелка (->).

Арифметика указателей на объект аналогична арифметике указателей на данные любого другого типа: она выполняется относительно объекта. Например, если указатель на объект инкрементируется, то он начинает указывать на следующий объект. Если указатель на объект декрементируется, то он начинает указывать на предыдущий объект.

Примеры

1. Пример арифметики указателей на объекты:

```
// Указатели на объекты
#include <iostream>
using namespace std;

class samp {
   int a, b;
public:
   samp(int n, int m) { a = n; b = m; }
   int get_a() { return a; }
   int get_b() { return b; }
};

int main()
{
   samp ob[4] = {
      samp(1, 2),
      samp(3, 4),
      samp(5, 6),
      samp(7, 8)
    };
   int i;

   samp *p;

   p = ob; // получение адреса начала массива

   for(i=0; i<4; i++) {
      cout << p->get_a() << ' ';
      cout << p->get_b() << "\n";
      p++; // переход к следующему объекту
   }

   cout << "\n";

   return 0;
}
```

Эта программа выводит на экран следующее:

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

Как видно из результата, при каждом инкрементировании указателя **p** он указывает на следующий объект массива.

4.3. Указатель *this*

C++ содержит специальный указатель **this.** Это указатель, который автоматически передается любой функции-члену при ее вызове и указывает на объект, генерирующий вызов. Например, рассмотрим следующую инструкцию:

```
ob.f1(); //предположим, что ob  - это объект
```

Функции **f1()** автоматически передается указатель на объект **ob.** Этот указатель и называется **this.**

Важно понимать, что указатель **this** передается только функциям-членам. Дружественным функциям указатель **this** не передается.

Примеры

1. Как вы уже видели, если функция-член работает с другим членом того же класса, она делает это без уточнения имени класса или объекта. Например, исследуйте эту короткую программу, в которой создается простой класс **inventory:**

```
// Демонстрация указателя this
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
   char item[20];
   double cost;
   int on_hand;
public:
   inventory(char *i, double c, int o)
   {
     strcpy(item, i);
     cost = c;
     on_hand = o;
   }
   void show();
};

void inventory::show()
{
   cout << item;
   cout << ": $" << cost;
   cout << "  On hand: " << on_hand << "\n";
}

int main()
{
```

```
   inventory ob("wrench", 4.95, 4);

   ob.show();

   return 0;
}
```

Обратите внимание, что внутри конструктора **inventory()** и функции-член **show()** переменные-члены **item, cost и on_hand** упоминаются явно. Так происходит потому, что функция-член может вызываться только в связи с объектом. Следовательно, в данном случае компилятор "знает", данные како! объекта имеются в виду.

Однако имеется еще более тонкое объяснение. Если вызывается функция-член, ей автоматически передается указатель **this** на объект, который является источником вызова. Таким образом, предыдущую программу можно переписать так:

```
// Демонстрация указателя this
#include <iostream>
#include <cstring>
using namespace std;

class inventory {
   char item[20];
   double cost;
   int on_hand;
public:
   inventory(char *i, double c, int o)
   {
      strcpy(this->item, i); // доступ к члену
      this->cost = c;        // через
      this->on_hand = o;     // указатель this
   }
   void show();
};

void inventory::show()
{
   cout << this->item; // использование this для доступа к членам
   cout << ": $" << this->cost;
   cout << "  On hand: " << this->on_hand << "\n";
}

int main()
{
   inventory ob("wrench", 4.95, 4);

   ob.show();

   return 0;
}
```

Здесь к переменным-членам объекта **ob** осуществляется прямой доступ через указатель **this**. Таким образом, внутри функции **show()** следующие две инструкции равнозначны:

```
cost = 123.23;
this->cost = 123.23;
```

На самом деле первая форма — это сокращенная запись второй.

Пока, наверное, еще не родился программист C++, который бы использовал указатель **this** для доступа к членам класса так, как было показано, поскольку сокращенная форма намного проще, но здесь важно понимать, что под этим сокращением подразумевается.

Использовать указатель **this** можно по-разному. Он особенно полезен при перегрузке операторов. Такое его применение более подробно будет изучаться в главе 6. На данный момент важно то, что по умолчанию всем функциям-членам автоматически передается указатель на вызывающий объект.

4.4. Операторы ***new*** и ***delete***

До сих пор при выделении динамической памяти вы использовали функцию **mallocO,** а при освобождении памяти — функцию **free().** Вместо этих стандартных функций в C++ стал применяться более безопасный и удобный способ выделения и освобождения памяти. Выделить память можно с помощью оператора **new,** а освободить ее с помощью оператора **delete.** Ниже представлена основная форма этих операторов:

```
p-var = new type;
delete p-var;
```

Здесь ***type*** —• это спецификатор типа объекта, для которого вы хотите выделить память, *а.* ***p-var*** — указатель на этот тип. **New** — это оператор, который возвращает указатель на динамически выделяемую память, достаточную для хранения объекта типа ***type***. Оператор **delete** освобождает эту память, когда в ней отпадает необходимость. Вызов оператора **delete** с неправильным указателем может привести к разрушению системы динамического выделения памяти и возможному краху программы.

Если свободной памяти недостаточно для выполнения запроса, произойдет одно из двух: либо оператор **new** возвратит нулевой указатель, либо будет сгенерирована исключительная ситуация. (Исключительные ситуации и обработка исключительных ситуаций описываются далее в этой книге. Коротко об исключительной ситуации можно сказать следующее — это динамическая ошибка, которую можно обработать определенным образом.) В соответствии с требованиями языка Standart C++ по умолчанию оператор **new** должен генерировать исключительную ситуацию при невозможности удовлетворить запрос на выделение памяти. Если ваша программа не обрабатывает эту исключительную ситуацию, выполнение программы прекращается. К сожалению, точные требования к тому, какие действия должны выполняться, если оператор **new** не в состоянии удовлетворить запрос на выделение памяти, за последние годы менялись несколько раз. Поэтому вполне возможно, что в вашем компиляторе реализация оператора **new** выполнена не так, как это предписано стандартом Standart C++.

Когда C++ только появился, при невозможности удовлетворить запрос на выделение памяти оператор **new** возвращал нулевой указатель. В дальнейшем эта ситуация изменилась, и при неудачной попытке выделения памяти оператор **new** стал генерировать исключительную ситуацию. В конце концов было принято решение, что при неудачной попытке выделения памяти оператор **new** будет генерировать исключительную ситуацию по умолчанию, а возвращение нулевого указателя останется в качестве возможной опции. Таким образом, реализация оператора **new** оказалась разной у разных производителей компиляторов. К примеру, во время написания этой книги в компиляторе Microsoft Visual C++ при невозможности удовлетворить запрос на выделение памяти оператор **new**

возвращал нулевой указатель, а в компиляторе Borland C++ генерировал исключительную ситуацию. Хотя в будущем во всех компиляторах оператор **new** будет реализован в соответствии со стандартом Standart C++, в настоящее время единственным способом узнать, какие именно действия он выполняет при неудачной попытке выделить память, является чтение документации на компилятор.

Поскольку имеется два возможных способа, которыми оператор **new** может сигнализировать об ошибке выделения памяти, и поскольку в разных компиляторах он может быть реализован по-разному, в примерах программ этой книги сделана попытка удовлетворить оба требования. Во всех примерах значение возвращаемого оператором **new** указателя проверяется на равенство нулю. Такое значение указателя обрабатывается теми компиляторами, в которых при неудачной попытке выделить память оператор **new** возвращает нуль, хотя никак не влияет на те, в которых оператор **new** генерирует исключительную ситуацию.' Если в вашем компиляторе во время выполнения программы при неудачной попытке выделить память оператор **new** сгенерирует исключительную ситуацию, то такая программа просто завершится. В дальнейшем, когда вы ближе познакомитесь с обработкой исключительных ситуаций, мы вернемся к оператору **new,** и вы узнаете, как лучше обрабатывать неудачные попытки выделения памяти. Вы также узнаете об альтернативной форме оператора **new,** который при наличии ошибки всегда возвращает нулевой указатель.

И последнее замечание: ни один из примеров программ этой книги не должен вести к ошибке выделения памяти при выполнении оператора **new,** поскольку в каждой конкретной программе выделяется лишь считанное число байтов.

Хотя операторы **new и delete** выполняют сходные с функциями **malloc()** и **free()** задачи, они имеют несколько преимуществ перед ними. Во-первых, оператор **new** автоматически выделяет требуемое для хранения объекта заданного типа количество памяти. Вам теперь не нужно использовать **sizeof,** например, для, подсчета требуемого числа байтов. Это уменьшает вероятность ошибки. Во-вторых, оператор **new** автоматически возвращает указатель на заданный тип данных. Вам не нужно выполнять приведение типов, операцию, которую вы делали, когда выделяли память, с помощью функции **malloc()** (см. следующее замечание). В-третьих, как оператор **new,** так и оператор **delete** можно перегружать, что дает возможность простой реализации вашей собственной, привычной модели распределения памяти. В-четвертых, Допускается инициализация объекта, для которого динамически выделена память. Наконец, больше нет необходимости включать в ваши программы заголовок **<cstdlib>.**

Теперь, после введения операторов **new и delete,** они будут использоваться программах вместо функций **malloc() и free().**

Примеры

1. Для начала рассмотрим программу выделения памяти для хранения целого.

```
// Простой пример операторов new и delete
#include <iostream>
using namespace std;

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

```
   p = new int; // выделение памяти для целого
if(!p) {
      cout << "Ошибка выделения памяти\n";
      return 1;
   }

   *p = 1000;

   cout << "Это целое, на которое указывает p: " << *p << "\n";

   delete p; // освобождение памяти

   return 0;
}
```

Обратите внимание, что возвращаемое оператором **new** значение перед использованием проверяется. Как уже упоминалось, эта проверка имеет значение только в том случае, если в вашем компиляторе при неудачной попытке выделения памяти оператор **new** возвращает нулевой указатель.

2. Пример, в котором память объекту выделяется динамически:

```
// Динамическое выделение памяти объектам.
#include <iostream>
using namespace std;

class samp {
   int i, j;
public:
   void set_ij(int a, int b) { i = a; j = b; }
   int get_product() { return i*j; }
};

int main()
{
   samp *p;

   p = new samp; // выделение памяти объекту
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      return 1;
   }

   p->set_ij( 4, 5);

   cout << "Итог равен:" << p->get_product() << "\n";

   return 0;
}
```

4.5. Дополнительные сведения об операторах ***new*** и ***delete***

В этом разделе обсуждаются два дополнительных свойства операторов **new** и **delete**. Во-первых, динамически размещаемому объекту может быть присвоено начальное значение. Во-вторых, можно создавать динамически размещаемые массивы объектов.

Вы можете присвоить динамически размещаемому объекту начальное значение,

используя следующую форму оператора **new:**

p-var = new *type* (*начальное_значение*);

Для динамически размещаемого одномерного массива используйте такую форму оператора **new:**

p-var = new *type [size];*

После выполнения оператора **p-var** будет указывать на начальный элемент массива из ***size*** элементов заданного типа. Из-за разных чисто технических причин невозможно инициализировать массив, память для которого выделена динамически.

Для удаления динамически размещённого одномерного массива вам следует использовать следующую форму оператора **delete:**

delete [] p-var;

При таком синтаксисе компилятор вызывает деструктор для каждого элемента массива. Это не приводит к многократному освобождению памяти по адресу, обозначенному указателем **p-var**. Она освобождается только один раз.

Примеры

1. В следующей программе выделяется и инициализируется память для хранения целого:

```
// Пример инициализации динамической переменной.
#include <iostream>
using namespace std;

int main()
{
   int *p;

   p = new int(9); // задание начального значения равного 9

   if(!p) {
      cout << "Ошибка выделения памяти\n";
      return 1;
   }

   cout << "Это целое, на которое указывает p: " << *p << "\n";

   delete p; // освобождение памяти

   return 0;
}
```

Как и следовало ожидать, программа выводит на экран число 9, являющееся начальным значением переменной, на которую указывает указатель **p.**

2. Следующая программа инициализирует динамически размещаемый объект:

```
// Динамическое выделение памяти объектам.
#include <iostream>
```

```
using namespace std;

class samp {
   int i, j;
public:
   samp(int a, int b) { i = a; j = b; }
   int get_product() { return i*j; }
};

int main()
{
   samp *p;

   p = new samp(6, 5); // размещение объекта с инициализацией
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      return 1;
   }

   cout << "Итог равен:" << p->get_product() << "\n";

   delete p;

   return 0;
}
```

При размещении объекта **samp** автоматически вызывается его конструктор, и объекту передаются значения 6 и 5.

3. В следующей программе размещается массив целых:

```
// Простой пример использования операторов new и delete.
#include <iostream>
using namespace std;

int main()
{
   int *p;

   p = new int [5]; // выделение памяти для 5 целых

   // Убедитесь, что память выделена
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      return 1;
   }

   int i;

   for(i=0; i<5; i++) p[i] = i;

   for(i=0; i<5; i++) {
      cout << "Это целое, на которое указывает p[" << i << "]: ";
      cout << p[i] << "\n";
  }

   delete [] p; // освобождение памяти

   return 0;
}
```

Эта программа выводит на экран следующее:

```
Это целое, на которое указывает p[0]: 0
Это целое, на которое указывает p[0]: 1
Это целое, на которое указывает p[0]: 2
Это целое, на которое указывает p[0]: 3
Это целое, на которое указывает p[0]: 4
```

4. В следующей программе создаётся динамический массив объектов:

```
// Динамическое выделение памяти для массива объектов.
#include <iostream>
using namespace std;

class samp {
   int i, j;
public:
   void set_ij(int a, int b) { i = a; j = b; }
   int get_product() { return i*j; }
};

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

   p = new samp [10]; // размещение массива объектов
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      return 1;
   }

   for(i=0; i<10; i++)
      p[i].set_ij(i, i);

   for(i=0; i<10; i++) {
      cout << "Содержимое [" << i << "] равно:" ;
      cout << p[i].get_product() << "\n";
   }
   delete [] p;
   return 0;
}
```

Эта программа выводит на экран следующее:

```
Содержимое [0] равно: 0
Содержимое [0] равно: 1
Содержимое [0] равно: 4
Содержимое [0] равно: 9
Содержимое [0] равно: 16
Содержимое [0] равно: 25
Содержимое [0] равно: 36
Содержимое [0] равно: 49
Содержимое [0] равно: 64
Содержимое [0] равно: 81
```

5. В новой версии предыдущей программы в неё вводится деструктор **samp** и теперь при освобождении памяти, обозначенной **p**, для каждого элемента массива вызывается деструктор:

```
// Динамическое выделение памяти для массива объектов.
#include <iostream>
using namespace std;
```

```
class samp {
   int i, j;
public:
   void set_ij(int a, int b) { i = a; j = b; }
   ~samp() { cout << "Удаление объекта...\n"; }
   int get_product() { return i*j; }
};

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

   p = new samp [10]; // размещение массива объектов
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      return 1;
   }

   for(i=0; i<10; i++)
      p[i].set_ij(i, i);

   for(i=0; i<10; i++) {
      cout << "Содержимое [" << i << "]  равно:" ;
      cout << p[i].get_product() << "\n";
   }
   delete [] p;
   return 0;
}
```

Эта программа выводит на экран следующее:

```
Содержимое [0] равно: 0
Содержимое [0] равно: 1
Содержимое [0] равно: 4
Содержимое [0] равно: 9
Содержимое [0] равно: 16
Содержимое [0] равно: 25
Содержимое [0] равно: 36
Содержимое [0] равно: 49
Содержимое [0] равно: 64
Содержимое [0] равно: 81
Удаление объекта…
Удаление объекта…
Удаление объекта…
Удаление объекта…
Удаление объекта…
Удаление объекта…
Удаление объекта…
Удаление объекта…
Удаление объекта…
Удаление объекта…
```

Как видите, деструктор **samp** вызывается десять раз – по разу на каждый элемент массива.

4.6. Ссылки

В C++ есть элемент, родственный указателю — это *ссылка (reference).* Ссылка является

скрытым указателем и во всех случаях, и для любых целей ее можно употреблять просто как еще одно имя переменной. Ссылку допустимо использовать тремя способами. Во-первых, ссылку можно передать в функцию. Во-вторых, ссылку можно возвратить из функции. Наконец, можно создать независимую ссылку. В книге рассмотрены все эти применения ссылки, начиная со ссылки в качестве параметра функции.

Несомненно, наиболее важное применение ссылки — это передача ее в качестве параметра функции. Чтобы помочь вам разобраться в том, что такое параметр-ссылка и как он работает, начнем с программы, в которой параметром является указатель (а не ссылка):

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

void f(int *n) ; //использование параметра-указателя
int main()
 {
        int i = 0;
       f(&i);
        cout « "Новое значение i: " « i « 'n\';
        return 0;
 }
void f(int *n)
{
      *n = 100; //занесение числа 100 в аргумент
                  //на который указывает указатель n }
}
```

Здесь функция **f()** загружает целое значение 100 по адресу, который обозначается указателем **n**. В данной программе функция **f()** вызывается из функции **main()** с адресом переменной **i**. Таким образом, после выполнения функции **f()** переменная **i** будет содержать число 100.

В этой программе показано, как использовать указатель для реализации механизма передачи параметра посредством вызова по ссылке (call by reference). В программах С такой механизм является единственным способом добиться вызова функции по ссылке. Однако в С++ с помощью параметра-ссылки можно полностью автоматизировать весь процесс. Чтобы узнать, как это сделать, изменим предыдущую программу. В ее новой вереи* используется параметр-ссылка:

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

void f(int &n); // объявление параметра-ссылки

 int main()
{
        int i = 0;

        f(i);

        cout « "Новое значение i: " « i « 'n\';

        return 0;
}
   // Теперь в функции f() используется параметр-ссылка
void f(int &n)
{
```

```
// отметьте, что в следующей инструкции знак * не требуется
n = 100; // занесение числа 100 в аргумент,
// используемый при вызове функции f()
}
```

Тщательно проанализируйте эту программу. Во-первых, для объявления параметра-ссылки перед именем переменной ставится знак амперсанда (&). Таким образом, переменная n объявляется параметром функции **f().** Теперь, поскольку переменная n является ссылкой, больше не нужно и даже неверно указывать оператор *. Вместо него всякий раз, когда переменная n упоминается внутри функции **f(),** она автоматически трактуется как указатель на аргумент, используемый при вызове функции **f().** Это значит, что инструкция

```
n = 100;
```

фактически помещает число 100 в переменную, используемую при вызове функции **f(),** каковой в данном случае является переменная i. Далее, при вызове функции **f()** перед аргументом не нужно ставить знак &. Вместо этого, поскольку функция **f()** объявлена как получающая параметр-ссылку, ей *автоматически* передается адрес аргумента.

Повторим, при использовании параметра-ссылки компилятор автоматически передает функции адрес переменной, указанной в качестве аргумента. Нет необходимости (а на самом деле и не допускается) получать адрес аргумента с помощью знака &. Более того, внутри функции компилятор автоматически использует переменную, на которую указывает параметр-ссылка. Нет необходимости (и опять не допускается) ставить знак *. Таким образом, параметр-ссылка полностью автоматизирует механизм передача диметра посредством вызова функции по ссылке.

Важно понимать следующее: адрес, на который указывает ссылка,| изменить не можете. Например, если в предыдущей программе инструкция

```
n++;
```

находилась бы внутри функции **f(),** ссылка **n** по-прежнему указывала бы на переменную **i** в функции **main().** Вместо инкрементирования адреса на который указывает ссылка **n,** эта инструкция инкрементирует значение переменной (в данном случае это переменная **i**).

Параметры-ссылки имеют несколько преимуществ по сравнению аналогичными (более или менее) альтернативными параметрами-углями. Во-первых, с практической точки зрения нет необходимости получать и передавать в функцию адрес аргумента. При использовании параметра-ссылки адрес передается автоматически. Во-вторых, по мнению многих программистов, параметры-ссылки предлагают более понятный и элегантный интерфейс, чем неуклюжий механизм указателей. В-третьих вы увидите в следующем разделе, при передаче объекта функции через ссылку копия объекта не создается. Это уменьшает вероятность ошибок данных с построением копии аргумента и вызовом ее деструктора.

Примеры

1. Классическим примером передачи аргументов по ссылке является функция, меняющая местами значения двух своих аргументов. В данном примере в функции **swapargs()** ссылки используются для того, чтобы поменять местами два ее целых аргумента:

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

void swapargs(int &x, int &y);

int main()
{
   int i, j;

   i = 10;
   j = 19;

   cout << "i: " << i << ", ";
   cout << "j: " << j << "\n";

   swapargs(i, j);

   cout << "После перестановки: ";
   cout << "i: " << i << ", ";
   cout << "j: " << j << "\n";

   return 0;
}

void swapargs(int &x, int &y)
{
   int t;

   t = x;
   x = y;
   y = t;
}
```

Если бы при написании функции **swapargs()** вместо ссылок использовались указатели. То функция выглядела бы следующим образом:

```
void swapargs(int *x, int *y)
{
   int t;

   t = *x;
   *x = *y;
   *y = t;
}
```

Как видите, благодаря использованию ссылок в функции **swapargs()**. Отпадает необходимость указывать знак *.

2. В следующей программе с помощью функции **round()** округляется значение типа **double**. Округляемое значение передаётся по ссылке.

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

void round(double &num);

int main()
{
   double i = 100.4;
```

```
  cout << i << "после округления ";
  round(i);
  cout << i << "\n";

  i = 10.9;
  cout << i << "после округления ";
  round(i);
  cout << i << "\n";

  return 0;
}

void round(double &num)
{
  double frac;
  double val;

  // разложение num на целую и дробную части
  frac = modf(num, &val);

  if(frac < 0.5) num = val;
  else num = val + 1.0;
}
```

В функции **round()** для разложения числа на целую и дриобную части указана редко используемая функция **modf()**. Возвращаемым значением этой функции является дробная часть, целая часть помещается в переменную, на которую указывает второй параметр функции **modf()**.

4.7. Передача ссылок на объекты

Как вы узнали из главы 3, если объект передается функции по значению,то в функции создается его копия. Хотя конструктор копии не вызывается, при возвращении функцией своего значения вызывается деструктор копии. Повторный вызов деструктора может привести в некоторых случаях к серьезным проблемам (например, при освобождении деструктором динамической памяти). I

Решает эту проблему передача объекта по ссылке. (Другое решение, о котором будет рассказано в главе 5, подразумевает использование конструктора копий.) При этом копия объекта не создается, и поэтому при возвращении функцией своего значения деструктор не вызывается. Тем не менее, запомните: изменения объекта внутри функции влияют на исходный объект, указанный в качестве аргумента функции.

Примеры

1. В следующем примере на экран выводится значение, передаваемое объекту по ссылке, но сначала рассмотрим версию программы, в которой объект типа **myclass()** передается в функцию **f()** по значению:

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

class myclass {
  int who;
public:
```

```
  myclass(int n) {
    who = n;
    cout << "Работа конструктора " << who << "\n";
  }
  ~myclass() { cout << "Работа деструктора " << who << "\n"; }
  int id() { return who; }
};

// o передается по значению
void f(myclass o)
{
  cout << "Получено" << o.id() << "\n";
}

int main()
{
  myclass x(1);

  f(x);

  return 0;
}
```

Эта функция выводит на экран следующее:

```
Работа конструктора 1
Получено 1
Работа деструктора, 1
Работа деструктора 1
```

Как видите, деструктор вызывается дважды: первый раз, когда после выполнения функции **f()** удаляется копия объекта 1, а второй раз — по окончании программы.

С другой стороны, если изменить программу так, чтобы использовать параметр-ссылку, то копия объекта не создается и поэтому после выполнения функции **f()** деструктор не вызывается:

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

class myclass {
  int who;
public:
  myclass(int n) {
    who = n;
    cout << "Работа конструктора " << who << "\n";
  }
  ~myclass() { cout << "Работа деструктора " << who << "\n"; }
  int id (){ return who; }
};

// Теперь o передается по ссылке
void f(myclass &o)
{
  // отметьте, что по-прежнему используется оператор .
  cout << "Получено" << o.id() << "\n";
}

int main()
{
  myclass x(1);
```

```
  f(x);

  return 0;
}
```

Эта версия предыдущей программы выводит на экран следующее:

```
Работа конструктора 1
Получено 1
Работа деструктора 1
```

4.8. Ссылка в качестве возвращаемого значения функции

Функция может возвращать ссылку. Как вы увидите в главе 6, возвращение ссылки может оказаться полезным при перегрузке операторов определенных типов. Кроме этого возвращение ссылки позволяет использовать функцию слева в инструкции присваивания. Это приводит к важному и неожиданному результату.

Примеры

1. Для начала, простая программа с функцией, которая возвращает ссылку:

```
// Простой пример ссылки в качестве возвращаемого значения функции
#include <iostream>
using namespace std;

int &f();
int x;

int main()
{
  f() = 100; // присваивание 100 ссылке, возвращаемой функцией f()

  cout << x << "\n";

  return 0;
}

// Возвращение ссылки на целое
int &f()
{
  return x; // возвращает ссылку на x
}
```

Здесь функция **f()** объявляется возвращающей ссылку на целое. Внутри тела функции инструкция

```
 return x;
```

не возвращает значение глобальной переменной **x,** она автоматически возвращает адрес переменной x (в виде ссылки). Таким образом, внутри функции **main()** инструкция

```
f() = 100;
```

заносит значение 100 в переменную **x,** поскольку функция f() уже возвратила ссылку на нее.

Повторим, функция **f()** возвращает ссылку. Когда функция **f()** указана слева в инструкции присваивания, то таким образом слева оказывается ссылка на объект, которую возвращает эта функция. Поскольку функция **f()** возвращает ссылку на переменную x (в данном примере), то эта переменная x и получает значение 100.

2. Вам следует быть внимательными при возвращении ссылок, чтобы объект, на который вы ссылаетесь, не вышел из области видимости. Например, рассмотрим эту, слегка переделанную функцию **f():**

```
// Возвращение ссылки на целое
int &f()
{
        int x // x — локальная переменная
        return x; // возвращение ссылки на x
}
```

В этом случае x становится локальной переменной функции **f()** и выходит из области видимости после выполнения функции. Это означает, что ссылку, возвращаемую функцией **f(),** уже нельзя использовать.

3. В качестве возвращаемого значения функции ссылка может оказаться полезной при создании массива определенного типа — так называемого защищенного массива (bounded array). Как вы знаете, в C и C++ контроль границ массива не производится. Следовательно, имеется вероятность при заполнении массива выйти за его границы. Однако в C++ можно создать класс массива с автоматическим контролем границ (automatic bounds checking). Любой класс массива содержит две основные функции — одну для запоминания информации в массиве и другую для извлечения информации. Именно эти функции в процессе работы могут проверять, не нарушены ли границы массива.

Следующая программа реализует контроль границ символьного массива:

```
// Пример защищенного массива
#include <iostream>
#include <cstdlib>
using namespace std;

class array {
   int size;
   char *p;
public:
   array(int num);
   ~ array() { delete [] p; }
   char &put(int i);
   char get(int i);
};

array::array(int num)
{
   p = new char [num];
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      exit(1);
   }
```

```
  size = num;
}

// Заполнение массива
char &array::put(int i)
{
  if(i<0 || i>=size) {
    cout << "Ошибка, нарушены границы массива!!!\n";
    exit(1);
  }
  return p[i]; // возврат ссылки на p[i]
}

// Получение чего-нибудь из массива
char array::get(int i)
{
  if(i<0 || i>=size) {
    cout << "Ошибка, нарушены границы массива!!!\n";
    exit(1);
  }
  return p[i]; // символ возврата
}

int main()
{
  array a(10);

  a.put(3) = 'X';
  a.put(2) = 'R';

  cout << a.get(3) << a.get(2) ;
  cout << "\n";

  /* теперь генерируем динамическую ошибку, связанную с нарушением границ массива */
  a.put(11) = '!';

  return 0;
}
```

Это был пример практического применения ссылок в качестве возвращаемого значения функций, и вам следует тщательно его изучить. Отметьте, что функция **put()** возвращает ссылку на элемент массива, заданный параметром i. Если индекс, заданный параметром **i,** не выходит за границы массива, та чтобы поместить значение в массив, эту ссылку можно использовать слева н инструкции присваивания. Обратной функцией является функция **get(),** которая возвращает заполненное по заданному индексу значение, если этот иж деке находится внутри диапазона. При таком подходе к работе с массивом он иногда упоминается как *безопасный массив (safe array).*

Имеется еще одна вещь, которую следует отметить в предыдущей программме, — это использование оператора **new** для динамического выделения памяти. Этот оператор дает возможность объявлять массивы различной длины.

Как уже упоминалось, способ контроля границ массива, реализованный в программе, является примером практического применения С++. Если вам необходимо во время работы программы проверять границы массива, такой способ позволяет вам легко этого добиться. Тем не менее, запомните: контроль границ замедляет доступ к массиву. Поэтому контроль границ лучше включать в программу только в том случае, если имеется высокая степень вероятности нарушения границ массива.

4.9. Независимые ссылки и ограничения на применение ссылок

Хотя они обычно и не используются, вы можете создавать *независимые ссылки (independent reference).* Независимая ссылка — это ссылка, которая во всех случаях является просто другим именем переменной. Поскольку ссылкам нельзя присваивать новые значения, независимая ссылка должна быть инициализирована при ее объявлении.

Имеется несколько ограничений, которые относятся к ссылкам всех типов. нельзя ссылаться на другую ссылку. Нельзя получить адрес ссылки. Нельзя создавать массивы ссылок и ссылаться на битовое поле. Ссылка должна быть инициализирована до того, как стать членом класса, возвратить значение функции или стать параметром функции.

Примеры

1. Пример программы с независимой ссылкой:

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

int main()
{
   int x;
   int &ref = x; // создание независимой ссылки

   x = 10;       // эти две инструкции
   ref = 10;     // функционально идентичны

   ref = 100;
   // здесь дважды печатается число 100
   cout << x << ' ' << ref <<"\n";

   return 0;
}
```

В этой программе независимая ссылка **ref** служит другим именем переменной **x**. С практической точки зрения **ref** и **x** идентичны.

2. Независимая ссылка может ссылаться на константу. Например, следующая инструкция вполне допустима:

```
const int &ref = 10;
```

В ссылках такого типа выгода невелика, но иногда их можно встретить в программах.