Глава 6
Введение в перегрузку операторов

В этой главе рассматривается очередное важное свойство C++: перегрузка операторов. Это свойство позволяет определять значение операторов C++ относительно задаваемых вами классов. Путём перегрузки связанных с классами операторов можно легко добавлять в программу новые типы данных.

6.1. Основы перегрузки операторов

Перегрузка операторов напоминает перегрузку функций. Более того, перегрузка операторов является фактически одним из видов перегрузки функций. Однако при этом вводятся некоторые дополнительные правила. Например, оператор всегда перегружается относительно определенного пользователем типа данных, такого, как класс. Другие отличия будут обсуждаться ниже по мере необходимости.

Когда оператор перегружается, то ничего из его исходного значения не теряется. Наоборот, он приобретает дополнительное значение, связанное с классом, для которого оператор был определен.

Для перегрузки оператора создается *оператор-функция (operator function)*. Чаще всего, оператор-функция является членом класса или дружественной классу, для которого она определена. Однако есть небольшая разница между оператор-функцией — членом класса и дружественной оператор-функцией. В первой части этой главы обсуждается создание оператор-функций — членов класса. О дружественных оператор-функциях будет рассказано далее в этой главе.

Здесь представлена основная форма оператор-функции — члена класса:

```
возвращаемый_тип имя_класса: :operator# (список аргументов)
{
    // выполняемая операция
}
```

Часто типом возвращаемого значения оператор-функции является класс, для которого она определена. (Хотя оператор-функция может возвращать данные любого типа.) В представленной общей форме оператор-функции вместо знака # нужно подставить перегружаемый оператор. Например, если перегружается оператор +, то у функции должно быть имя **operator+.** Содержание списка ***список-аргументов*** зависит от реализации оператор-функции п от типа перегружаемого оператора.

Следует запомнить два важных ограничения на перегрузку операторов. Во-первых, нельзя менять приоритет операторов. Во-вторых, нельзя менять число операндов оператора. Например, нельзя перегрузить оператор / так, чтобы в нем использовался только один операнд.

Большинство операторов C++ можно перегружать. Ниже представлены те несколько операторов, которые перегружать нельзя:

. :: .* ?

Кроме того, нельзя перегружать операторы препроцессора. (Оператор .* является сугубо специальным и в книге не рассматривается.)

Запомните, что в C++ понятие оператора трактуется очень широко: в это понятие входят оператор индексирования **[],** оператор вызова функции **(),** операторы **new и delete,** операторы . (точка) и -> (стрелка). Однако в данной главе мы коснемся более обычных операторов.

Оператор-функции, за исключением оператора **=,** наследуются производным классом. Тем не менее для производного класса тоже можно перегрузить любой выбранный оператор (включая операторы, уже перегруженные в базовом классе).

Вы уже пользовались двумя перегруженными операторами: << и >>, которые перегружались для реализации ввода/вывода. Как уже упоминалось, перегрузка этих операторов для реализации ввода/вывода не мешает им выполнять свои традиционные функции левого и правого сдвига.

Хотя допустимо иметь оператор-функцию для реализации *любого* действия — связанного или нет с традиционным употреблением оператора — лучше, если действия перегружаемых операторов остаются в сфере их традиционного использования. При создании перегружаемых операторов, для которых этот принцип не поддерживается, имеется риск существенного снижения читабельности программ. Например, перегрузка оператора / так, чтобы 300 раз записать в дисковый файл фразу "Мне нравится C++", является явным злоупотреблением перегрузкой операторов.

Несмотря на вышесказанное, иногда может потребоваться использовать какой-либо оператор нетрадиционным образом. Типичным примером этого как Раз и являются перегруженные для ввода/вывода операторы << и >>. Однако Даже в этом случае, левые и правые стрелки обеспечивают визуально понятный смысл их значения. Поэтому, даже если вам очень хочется перегрузить какой-нибудь оператор нестандартным способом, лучше приложите дополнительные усилия и постарайтесь воспользоваться .каким-нибудь более подходящим оператором.

И последнее, оператор-функции не могут иметь параметров по умолчанию.

6.2. Перегрузка бинарных операторов

Когда оператор-функция — член класса перегружает бинарный оператор, у функции будет только один параметр. Этот параметр получит тот объект, который расположен справа от оператора. Объект слева генерирует вызов оператор-функции и передается неявно, с помощью указателя **this.**

Важно понимать, что для написания оператор-функций имеется множество вариантов. Примеры, показанные здесь и в других местах главы, не являются исчерпывающими, хотя

они иллюстрируют несколько наиболее общих технических приемов.

Примеры

1. В следующей программе перегружается оператор + относительно класса **coord.** Этот класс используется для поддержания координат X,Y.

```
// Перегрузка оператора + относительно класса coord
#include <iostream>
using namespace std;

class coord {
   int x,y; // значения координат
public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
   void get_xy(int &i, int &j) { i = x; j = y; }
   coord operator+(coord ob2);
};

// Перегрузка оператора + относительно класса coord
coord coord::operator+(coord ob2)
{
   coord temp;
   temp.x = x +ob2.x;
   temp.y = y +ob2.y;

   return temp;
}

int main()
{
   coord o1(10, 10), o2(5, 3), o3;
   int x, y;

   o3 = o1 + o2; // сложение двух объектов - вызов функции operator+()

   o3.get_xy(x, y);
   cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

   return 0;
}
```

После выполнения программы на экран выводится следующее:

(Ol + 02) X: 15, Y: 13

Давайте внимательно рассмотрим программу. Функция **operator+0** возвращает объект типа **coord,** в котором сумма координат по оси X находится в переменной **x,** а сумма координат по оси Y — в переменной y. Отметьте, что временный объект **temp** используется внутри функции **operator+0** для хранения результата и является возвращаемым объектом. Отметьте также, что ни один из операндов не меняется. Назначение переменной **temp** легко понять.

В данной ситуации (как и в большинстве ситуаций) оператор + был перегружен способом, аналогичным своему традиционному арифметическому использованию. Поэтому и было важно, чтобы ни один из операндов не менялся. Например, когда вы складываете 10+4,

результат равен 14, но ни 10, ни 4 не меняются. Таким образом, временный объект необходим для хранения результата.
Смысл того, что функция **operator+0** возвращает объект типа **coord,** состоит в том, что это позволяет использовать результат сложения объектов типа **coord** в сложном выражении. Например, инструкция

```
o3 =  ol + o2;
```

правильна только потому, что результат выполнения операции **ol + o2** является объектом, который можно присвоить объекту **o3.** Если бы возвращаемым значением был объект другого типа, то эта инструкция была бы неправильна- Более того, возвращая объект типа **coord,** оператор сложения допускает возможность существования строки, состоящей из нескольких сложений. Например, следующая инструкция вполне корректна:

```
o3=  o1 + o2 + o1 +o3;
```

Хотя у вас будут ситуации, в которых понадобится оператор-функция, возвращающая нечто иное, чем объект класса, для которого она определена, большинство создаваемых вами оператор-функций будут возвращать именно такие объекты. (Основное исключение из этого правила связано с перегрузкой операторов отношения и логических операторов. Эта ситуация исследуется в разделе 6.3 "Перегрузка операторов отношения и логических операторов" далее в этой главе.)

Последнее замечание по этому примеру. Поскольку объект типа **coord** является возвращаемым значением оператор-функции, то следующая инструкция также совершенно правильна:

```
(ol + o2).get_xy(x, y);
```

Здесь временный объект, возвращаемый функцией **operator+(),** используется непосредственно. Естественно, что-после выполнения этой инструкции временный объект удаляется.

2. В следующей версии предыдущей программы относительно класса **coord** перегружаются операторы — и **=.**

```
// Перегрузка операторов +, - и = относительно класса coord
#include <iostream>
using namespace std;

class coord {
   int x, y; // значения координат
public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
   void get_xy(int &i, int &j) { i = x; j = y; }
   coord operator+(coord ob2);
   coord operator-(coord ob2);
   coord operator=(coord ob2);
};

// Перегрузка оператора + относительно класса coord
coord coord::operator+(coord ob2)
{
   coord temp;
```

```
    temp.x = x + ob2.x;
    temp.y = y + ob2.y;

    return temp;
  }

  // Перегрузка оператора - относительно класса coord
  coord coord::operator-(coord ob2)
  {
    coord temp;

    temp.x = x - ob2.x;
    temp.y = y - ob2.y;

    return temp;
  }

  // Перегрузка оператора = относительно класса coord
  coord coord::operator=(coord ob2)
  {
    x = ob2.x;
    y = ob2.y;

    return *this; // возвращение объекта, которому присвоено значение
  }

  int main()
  {
    coord o1(10, 10), o2(5, 3), o3;
    int x, y;

    o3 = o1 + o2; // сложение двух объектов - вызов функции operator+()
    o3.get_xy(x, y);
    cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1 - o2; // вычитание двух объектов - вызов функции operator-()
    o3.get_xy(x, y);
    cout << "(o1 - o2) X: " << x << ", Y: " << y << "\n";

    o3 = o1; // присваивание объекта- вызов функции operator=()
    o3.get_xy(x, y);
    cout << "(o3 = o1) X: " << x << ", Y: " << y << "\n";

    return 0;
}
```

Реализация функции **operator-()** похожа на реализацию функции **operator+().** Однако она иллюстрирует ту особенность перегрузки операторов, где важен порядок следования операндов. При создании функции **operator+()** порядок следования операндов значения не имел. (То есть A+B тождественно B+A.) Однако результат операции вычитания зависит от порядка следования операндов. Поэтому, чтобы правильно перегрузить оператор вычитания, необходимо вычесть правый операнд из левого. Поскольку левый операнд генерирует вызов функции **operator-0,** порядок вычитания должен быть следующим:

```
x - ob2.x;
```

Теперь рассмотрим оператор-функцию присваивания. В первую очередь необходимо отметить, что левый операнд (т. е. объект, которому присваивается значение) после выполнения операции меняется. Здесь сохраняется обычный смысл присваивания. Во-вторых, функция возвращает указатель ***this.** Это происходит потому, что функция **operator==()** возвращает тот объект, котороД му присваивается значение. Таким образом

удается выстраивать операторья присваивания в цепочки. Как вы уже должны знать, в C++ следующая инструкция синтаксически правильна (и на практике вполне обычна):

```
a = b = c = d = 0;
```

Возвращая указатель ***this,** перегруженный оператор присваивания дает возможность подобным образом выстраивать объекты типа **coord.** Например, представленная ниже инструкция вполне корректна^

```
o3 = o2 = o1;
```

Запомните, нет правила, требующего, чтобы перегруженная оператор-функция присваивания возвращала объект, полученный в результате присваивания. Однако если вы хотите перегрузить оператор = относительно класса, то, как и в случае присваивания встроенных типов данных, он должен возвращать указатель ***this.**

3. Имеется возможность перегрузить оператор относительно класса так, что правый операнд будет объектом встроенного типа, например, целого, а не объектом того класса, членом которого является оператор-функция. Например, в приведенном ниже примере оператор + перегружается так, что прибавляет целое значение к объекту типа **coord:**

```
// Перегрузка оператора + как для операции ob+ob,
// так и для операции ob+int
#include <iostream>
using namespace std;

class coord {
   int x, y; // значения координат
public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
   void get_xy(int &i, int &j) { i = x; j = y; }
   coord operator+(coord ob2); // ob + ob
   coord operator+(int i); // ob + int
};

// Перегрузка оператора + относительно класса coord
coord coord::operator+(coord ob2)
{
   coord temp;

   temp.x = x +ob2.x;
   temp.y = y +ob2.y;

   return temp;
}

// Перегрузка оператора + для операции ob+int
coord coord::operator+(int i)
{
   coord temp;

   temp.x = x +i;
   temp.y = y +i;

   return temp;
}

int main()
```

```
{
   coord o1(10, 10), o2(5, 3), o3;
   int x, y;

   o3 = o1 + o2; // сложение двух объектов
            // вызов функции operator+(coord)
   o3.get_xy(x, y);
   cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

   o3 = o1 + 100; // сложение объекта и целого
             // вызов функции operator+(int)
   o3.get_xy(x, y);
   cout << "(o1 + 100) X: " << x << ", Y: " << y << "\n";

   return 0;
}
```

Здесь важно запомнить следующее: когда оператор-функция — член класса перегружается так, чтобы объект этого класса можно было использовать в одной операции-с переменной встроенного типа данных, встроенный тип данных должен находиться справа от оператора. Смысл этого легко понять: он в том, что объект, который находится слева, генерирует вызов оператор-функции. Однако что произойдет, если компилятор встретит следующую инструкцию?

```
o3 = 19 + o1; // int + ob
```

Для обработки сложения целого с объектом встроенной операции не существует. Перегруженная функция **operator+(int i)** работает только в том случае, если объект находится слева от оператора. Поэтому эта инструкция приведет к ошибке при компиляции. (Позже вы узнаете способ обойти это ограничение.)

4. В оператор-функции можно использовать параметр-ссылку. Например, допустимым способом перегрузки оператора + относительно класса **coord** является следующий:

```
// Перегрузка + относительно класса coord, с использованием ссылки
coord coord::operator+(coord .&ob2)
{
        coord temp;
        temp.x = x + ob2.x;
        temp.y = y + ob2.y;

        return temp;
}
```

Одним из доводов в пользу использования ссылки в качестве параметра оператор-функции является ее эффективность. Передача объекта функции в качестве параметра часто требует больших затрат процессорного времени. Передача же адреса объекта всегда быстрее и эффективней. Если оператор многократно используется, параметр-ссылка обычно позволяет значительно повысить производительность.

Другой довод в пользу использования параметра-ссылки состоит в том, что ссылка позволяет избежать неприятностей, связанных с удалением копии операнда. Как вы знаете по предыдущим главам, при передаче аргумента по значению создается его копия. Если у такого объекта есть деструктор, то после завершения выполнения функции вызывается деструктор копии- Иногда возможны ситуации, когда деструктор удаляет нечто такое, что необходимо вызывающему объекту. Использование в этом случае в качестве параметра не самого объекта, а ссылки на этот объект -— это простой (и эффективный) способ избежать

проблем. Тем не менее, запомните, что в общем случае решить эту проблему могло бы определение конструктора копий.

6.3. Перегрузка операторов отношения и логических операторов

Существует возможность перегрузки операторов отношения и логических операторов. При перегрузке операторов отношения и логических операторов так, чтобы они вели себя обычным образом, не нужны оператор-функции, возвращающие объект класса, для которого эти оператор-функции определены. Вместо этого они должны возвращать целое, интерпретируемое как значение **true** или **false.** Помимо того, что возвращаемым значением таких оператор-функций должно ,быть значение **true** или **false,** должна быть возможность встраивания операторов отношения и логических операторов в большие выражения, включающие также данные других типов.

Примеры

1. В следующей программе перегружаются операторы **==** и ***&&****:*

```
// Перегрузка операторов == и && относительно класса coord
#include <iostream>
using namespace std;

class coord {
   int x, y; // значения координат
public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
   void get_xy(int &i, int &j) { i = x; j = y; }
   int operator==(coord ob2);
   int operator&&(coord ob2);
};

// Перегрузка оператора == для класса coord
int coord::operator==(coord ob2)
{
   return x==ob2.x && y==ob2.y;
}

// Перегрузка оператора && для класса coord
int coord::operator&&(coord ob2)
{
   return (x && ob2.x) && (y && ob2.y);
}

int main()
{
   coord o1(10, 10), o2(5, 3), o3(10, 10), o4(0, 0);

   if(o1==o2) cout << "o1 равно o2\n";
   else cout << "o1 не равно o2\n";

   if(o1==o3) cout << "o1 равно o3\n";
   else cout << "o1 не равно o3\n";

   if(o1&&o2) cout << "o1 && o2 равно истина\n";
   else cout << "o1 && o2 равно ложь\n";
```

```
  if(o1&&o4) cout << "o1 && o4 равно истина\n";
  else cout << "o1 && o4 равно ложь\n";
  return 0;
}
```

6.4. Перегрузка унарных операторов

Перегрузка унарных операторов аналогична перегрузке бинарных, за исключением того, что мы имеем дело не с двумя, а с одним операндом. При перегрузке унарного оператора с использованием функции-члена у функции нет параметров. Поскольку имеется только один операнд, он и генерируй вызов оператор-функции. Другие параметры не нужны.

Примеры

1. В следующей программе относительно класса **coord** перегружается оператор инкремента (++):

```
// Перегрузка оператора ++ относительно класса coord
#include <iostream>
using namespace std;

class coord {
  int x, y; // значения координат
public:
  coord() { x = 0; y= 0; }
  coord(int i, int j) { x = i; y = j; }
  void get_xy(int &i, int &j) { i = x; j = y; }
  coord operator++();
};

// Перегрузка оператора ++ для класса coord
coord coord::operator++()
{
  x++;
  y++;

  return *this;
}

int main()
{
  coord o1(10, 10);
  int x, y;

  ++o1; // инкремент объекта
  o1.get_xy(x, y);
  cout << "(++o1) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Поскольку оператор инкремента увеличивает свой операнд на единицу, перегрузка этого оператора меняет объект, с которым он работает. Это позволяет использовать оператор инкремента в качестве части более сложной инструкции, например, такой:

```
o2 = ++ol;
```

Как и в случае бинарных операторов, нет правила, которое заставляло бы перегружать

унарный оператор с сохранением его обычного смысла. Однако в большинстве случаев лучше поступать именно так.

2. В ранних версиях C++ при перегрузке оператора инкремента или декремента положения операторов ++ и — относительно операнда не различались. Поэтому по отношению к предыдущей программе следующие две инструкции эквивалентны:

```
o1++;
++o1 ;
```

Однако в современной спецификации C++ определен способ, по которому компилятор может различить эти две инструкции. В соответствии с этим способом задаются две версии функции **operator++().** Первая определяетеся так, как было показано в предыдущем примере. Вторая определяется следующим образом:

```
coord coord::operator++(int notused);
```

Если оператор ++ указан перед операндом, вызывается функция **operator++().** Если оператор ++ указан после операнда, вызывается функция **operator++(int notused).** В этом случае переменной **notused** передается значение 0. Таким образом, если префиксный и постфиксный инкремент или декремент важны для объектов вашего класса, то понадобится реализовать обе оператор-функции.

3. Как вы знаете, знак минус в C++ является как бинарным, так и унарным оператором. Вы, наверное, хотели бы знать, как его можно перегрузить относительно создаваемого вами класса так, чтобы оператор сохранил оба эти качества. Реальное решение достаточно элементарно: просто перегрузите его дважды, один раз как бинарный оператор, а второй — как унарный. Программа, реализующая этот прием, показана ниже:

```
// Перегрузка оператора - относительно класса coord
#include <iostream>
using namespace std;

class coord {
   int x, y; // значения координат
public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
   void get_xy(int &i, int &j) { i = x; j = y; }
   coord operator-(coord ob2); // бинарный минус
   coord operator-(); // унарный минус
};

// Перегрузка оператора - относительно класса coord
coord coord::operator-(coord ob2)
{
   coord temp;

   temp.x = x - ob2.x;
   temp.y = y - ob2.y;

   return temp;
}

// Перегрузка унарного оператора - для класса coord
coord coord::operator-()
```

```
{
   x = -x;
   y = -y;
   return *this;
}

int main()
{
   coord o1(10, 10), o2(5, 7);
   int x, y;

   o1 = o1 - o2; // вычитание
   o1.get_xy(x, y);
   cout << "(o1-o2) X: " << x << ", Y: " << y << "\n";

   o1 = -o1; // отрицание
   o1.get_xy(x, y);
   cout << "(-o1) X: " << x << ", Y: " << y << "\n";

   return 0;
}
```

Как видите, если минус перегружать как бинарный оператор, то у функции будет один параметр. Если его перегружать как унарный оператор, то параметров не будет. Это отличие в числе параметров и делает возможным перегрузку минуса для обоих операторов. Как показано в программе, при использовании минуса в качестве бинарного оператора вызывается функция **operator-(coord ob2),** а в качестве унарного — функция **operator-().**

6.5. Дружественные оператор-функции

Как отмечалось в начале этой главы, имеется возможность перегружать оператор относительно класса, используя не только функцию-член, но и дружественную функцию. Как вы знаете, дружественной функции указатель **this** Не передается. В случае бинарного оператора это означает, что дружественной оператор-функции явно передаются оба операнда, а в случае унарного — один. Все остальное в обоих случаях одинаково, и нет особого смысла вместо оператор-функции — члена класса использовать дружественную оператор-функцию, за одним важным исключением, которое будет рассмотрено в примерах.

Примеры

1. Здесь функция **operator+0** перегружается для класса **coord** с использованием дружственной функции:

```
// Перегрузка оператора + относительно класса coord
// с использованием дружественной функции
#include <iostream>
using namespace std;

class coord {
   int x, y; // значения координат
public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
   void get_xy(int &i, int &j) { i = x; j = y; }
   friend coord operator+(coord ob1, coord ob2);
```

```
};

// Перегрузка оператора + с использованием дружественной функции
coord operator+(coord ob1, coord ob2)
{
  coord temp;

  temp.x = ob1.x + ob2.x;
  temp.y = ob1.y + ob2.y;

  return temp;
}

int main()
{
  coord o1(10, 10), o2(5, 3), o3;
  int x, y;

  o3 = o1 + o2; // сложение двух объектов
          // вызов функции operator+()

  o3.get_xy(x, y);
  cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";

  return 0;
}
```

Обратите внимание, что левый операнд передается первому параметру, а правый—второму.

2. Перегрузка оператора посредством дружественной функции дает одну очень важную возможность, которой нет у функции — члена класса. Используя дружественную оператор-функцию, в операциях с объектами можно использовать встроенные типы данных, и при этом встроенный тип может распо-. лагаться слева от оператора. Как отмечалось ранее в этой главе, можно перегрузить оператор-функцию, являющуюся членом класса, так, что левый операнд становится объектом, а правый — значением встроенного типа. Но нельзя для функции -— члена класса располагать значение встроенного типа слева от оператора. Например, пусть перегружается оператор-функция --член класса, тогда первая показанная здесь инструкция правильна, а вторая нет:

```
ob1 = ob2 +10; // правильно
ob1 = 10 + ob2; // неправильно
```

Несмотря на то, что допустимо строить выражения так, как показано в первом примере, необходимость постоянно думать о том, чтобы объект находился слева от оператора, а значение встроенного типа — справа, может быть обременительной. Решение проблемы состоит в том, чтобы сделать перегруженную оператор-функцию дружественной и задать обе возможные ситуации.

Как вы знаете, дружественной оператор-функции передаются явно *оба* операнда. Таким образом, можно задать перегружаемую дружественную функцию так, чтобы левый операнд был объектом, а правый — операндом другого типа. Затем можно снова перегрузить оператор, чтобы левый операнд был значением встроенного типа, а правый — объектом- Следующая программа иллюстрирует такой подход:

```
// Дружественные оператор-функции придают гибкость программе
#include <iostream>
using namespace std;
```

```
class coord {
   int x, y; // значения координат
public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
   void get_xy(int &i, int &j) { i = x; j = y; }
   friend coord operator+(coord ob1, int i);
   friend coord operator+(int i, coord ob1);
};

// Перегрузка оператора + для операции ob + int
coord operator+(coord ob1, int i)
{
   coord temp;

   temp.x = ob1.x + i;
   temp.y = ob1.y + i;

   return temp;
}

// Перегрузка оператора + для операции int + ob
coord operator+(int i, coord ob1)
{
   coord temp;

   temp.x = ob1.x + i;
   temp.y = ob1.y + i;

   return temp;
}

int main()
{
   coord o1(10, 10);
   int x, y;

   o1 = o1 + 10; // объект + целое
   o1.get_xy(x, y);
   cout << "(o1 + 10) X: " << x << ", Y: " << y << "\n";

   o1 = 99 + o1; // целое + объект
   o1.get_xy(x, y);
   cout << "(99 + o1) X: " << x << ", Y: " << y << "\n";

   return 0;
}
```

В результате перегрузки дружественных оператор-функций становятся правильными обе инструкции:

```
ol = o1 + 10;
o1 = 99 + o1;
```

При использовании дружественной оператор-функции для перегрузки унарного оператора ++ или —— необходимо передавать операнд в функцию в качестве параметра-ссылки, поскольку дружественной функции не передается указатель **this**. Запомните, что в операторах инкремента и декремента подразумевается, что операнд будет изменен. Однако при перегрузке этих операторов посредством дружественных функций операнд передается по значению. Таким образом, любое изменение параметра внутри дружественной оператор-

функции не влияет на объект, являющийся источником вызова. Поскольку при использовании дружественной функции отсутствует явно передаваемый указатель на объект (т. е- указатель **this),** инкремент и декремент не влияют на операнд.

Однако при передаче операнда дружественной функции в качестве параметра-ссылки, изменения, которые имеют место внутри дружественной функции, влияют на объект, являющийся источником вызова. Например, в следующей программе посредством дружественной функции перегружается оператор ++.

```
// Перегрузка оператора ++ с использованием дружественной функции
#include <iostream>
using namespace std;

class coord {
   int x, y; // значения координат
public:
   coord() { x = 0; y= 0; }
   coord(int i, int j) { x = i; y = j; }
   void get_xy(int &i, int &j) { i = x; j = y; }
   friend coord operator++(coord &ob);
};

// Перегрузка оператора ++ с использованием дружественной функции
coord operator++(coord &ob) // использование ссылки
                    // в качестве параметра
{
   ob.x++;
   ob.y++;

   return ob; // возвращение объекта,
          // ставшего источником вызова
}

int main()
{
   coord o1(10, 10);
   int x, y;

   ++o1; // объект o1 передается по ссылке
   o1.get_xy(x, y);
   cout << "(++o1) X: " << x << ", Y: " << y << "\n";

   return 0;
}
```

Если вы используете современный компилятор, то с помощью дружественной оператор-функции можно определить разницу между префиксной и постфиксной формами операторов инкремента и декремента точно так же, как это делалось с помощью функций-членов. Просто добавьте целый параметр при задании постфиксной версии. Например, здесь приводятся префиксная и постфиксная версии оператора инкремента относительно класса **coord:**

```
coord operator++(coord &ob); // префиксная версия
coord operator++(coord Sob, int notused); // постфиксная версия
```

Если оператор ++ находится перед операндом, то вызывается функция **coord operator++(coord &ob).** Однако, если оператор ++ находится после операнда, вызывается функция **coord operator++(coord &ob, int notused). В** этом случае переменной **notused**

будет передано значение 0.

6.6. Особенности использования оператора присваивания

Как уже отмечалось, относительно класса можно перегрузить оператор присваивания. По умолчанию, если оператор присваивания применяется к объекту, то происходит поразрядное копирование объекта, стоящего справа от оператора, в объект, стоящий слева от оператора. Если это то, что вам нужно, нет смысла создавать собственную функцию **operator=().** Однако бывают случаи, когда точное поразрядное копирование нежелательно. В главе 3 при выделении памяти объекту вам было представлено несколько примеров подобного рода. В таких случаях требуется особая операция присваивания.

Примеры

1. Здесь приведена новая версия класса **strtype,** различные формы которого изучались в предыдущих главах. В этой версии оператор == перегружается так, что указатель p при присваивании не перезаписывается.

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

class strtype {
   char *p;
   int len;
public:
   strtype(char *s);
   ~strtype() {
      cout << "Освобождение памяти по адресу " << (unsigned) p << '\n';
      delete []p;
   }
   char *get() { return p; }
   strtype &operator=(strtype &ob);
};

strtype::strtype(char *s)
{
   int l;

   l = strlen(s) + 1;

   p = new char [l];
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      exit(1);
   }

   len = l;
   strcpy(p, s);
}

// Присваивание объекта
strtype &strtype::operator=(strtype &ob)
{
   // выяснение необходимости дополнительной памяти
   if(len < ob.len) { // требуется выделение дополнительной памяти
```

```
        delete[] p;
        p = new char [ob.len];
        if(!p) {
          cout << "Ошибка выделения памяти\n";
          exit(1);
        }
    }
    len = ob.len;
    strcpy(p, ob.p);
    return *this;
}

int main()
{
    strtype a("Привет"), b("Здесь");

    cout << a.get() << '\n';
    cout << b.get() << '\n';

    a = b; // теперь p не перезаписывается

    cout << a.get() << '\n';
    cout << b.get() << '\n';

    return 0;
}
```

Как видите, перегрузка оператора присваивания предотвращает перезапись указателя **p.** При первом контроле выясняется, достаточно ли в объекте слева от оператора присваивания выделено памяти для хранения присваиваемой ему строки. Если это не так, то память освобождается и выделяется новый фрагмент. Затем строка копируется в эту память, а длина строки копируется в переменную **len.**

Отметьте два важных свойства функции **operator=().** Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, стоящего справа от оператора присваивания- Как известно по предыдущим главам, при передаче в функцию объекта создается его копия, и эта копия удаляется при завершении работы функции- В этом случае для удаления копии должен вызываться деструктор, который освобождает память, обозначенную указателем р. Однако память по адресу р все еще необходима объекту, который является аргументом. Параметр-ссылка помогает решить проблему.

Вторым важным свойством функции **operator=()** является то, что она возвращает не объект, а ссылку на него. Смысл этого тот же, что и при обычном использовании параметра-ссылки. Функция возвращает временный объект, который удаляется после полного завершения ее работы- Это означает, что для временного объекта будет вызван деструктор, который вызовет освобождение памяти по адресу р, но указатель р (и память на которую он ссылается) все еще необходимы для присваивания значения объекту. Поэтому, чтобы избежать создания временного объекта, в качестве возвращаемого значения используется ссылка.

6.7. Перегрузка оператора индекса массива []

Последним оператором, который мы научимся перегружать, будет оператор индекса массива **[]**. В C++ при перегрузке оператор **[]** рассматривается как бинарный. Оператор **[]**

можно перегружать только как функцию-член. Ниже представлена основная форма оператор-функции — члена класса operator**[]()**:

тип ***имя_класса: :*** **operator** ***[]*** **(int** ***индекс*****)**

```
{
        //
        .
        .
        .
}
```

С технической точки зрения тип параметра не обязательно должен быть целым, но поскольку оператор-функция **operator []()**, как правило, используется для получения индекса массива, то ее параметр обычно имеет тип **int.**

Чтобы понять, как работает Оператор **[],** представим, что объект О индексируется следующим образом:

O[9]

Этот индекс транслируется в вызов функции **operator[]();**

O.operator[](9)

Таким образом, значение выражения внутри оператора индексирования явно передается функции **operator[]()** в качестве параметра. При этом указатель **this** будет ссылаться на объект **О,** являющийся источником вызова.

Примеры

1. В следующей программе объявляется состоящий из пяти целых массив **arraytype.** Каждый элемент массива инициализируется конструктором. Перегруженная функция **operator[]()** возвращает элемент, заданный ее параметром.

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

const int SIZE = 5;

class arraytype {
   int a[SIZE];
public:
   arraytype() {
      int i;
      for (i=o; i<SIZE; i++) a[i] = i;
   }
   int operator[](int i) { return a[i]; }
};

int main()
{
   arraytype ob;
   int i;

   for(i=0; i<SIZE; i++)
      cout << ob[i] << " ";
```

```
    return 0;
  }
```

В результате работы программы на экран выводится следующее:

```
  01234.
```

В этом и остальных примерах инициализация массива a с помощью конструктора выполнена исключительно в иллюстративных целях и на самом деле не требуется.

2. Имеется возможность перегрузить функцию **operator[]()** так, чтобы в инструкции присваивания оператор **[]** можно было располагать как слева, так и справа от оператора =. Для этого возвратите ссылку на индексируемый элемент. Следующая программа иллюстрирует такой подход.

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

const int SIZE = 5;

class arraytype {
  int a[SIZE];
public:
  arraytype() {
    int i;
    for (i=o; i<SIZE; i++) a[i] = i;
  }
  int &operator[](int i) { return a[i]; }
};

int main()
{
  arraytype ob;
  int i;

  for(i=0; i<SIZE; i++)
    cout << ob[i] << " ";

  cout << "\n";

  // добавление значения 10 к каждому элементу массива
  for(i=0; i<SIZE; i++)
    ob[i] = ob[i]+10; // оператор [] слева от оператора =

  for(i=0; i<SIZE; i++)
    cout << ob[i] << " ";

  return 0;
}
```

В результате работы программы на экран выводится следующее:

```
0 1 2 3 4
10 11 12 13 14
```

Поскольку теперь функция **operator[] ()** возвращает ссылку на элемент массива с индексом **i,** то для изменения этого элемента оператор **[]** можно расположить слева в инструкции присваивания. (Естественно, что как и прежде его можно располагать справа.) Таким образом, с объектами типа **arraytype** можно обращаться так же, как и с обычными

массивами.

3. Перегрузка оператора **[]** дает возможность по-новому взглянуть на задачу индексирования безопасного массива. Ранее в этой книге мы рассматривали простейший способ реализации безопасного массива, в котором для доступа к элементам массива использовались функции **get()** и **put().** Перегрузка оператора **[]** позволит нам теперь создать такой массив гораздо проще. Вспомните, что безопасный массив— это массив, который инкапсулирован в классе, и при этом класс обеспечивает контроль границ массива. Такой подход предотвращает нарушение границ массива. Благодаря перегрузке оператора **[]**, работать с безопасным массивом можно так же, как с обычным.

Для создания безопасного массива просто реализуйте в функции **operator[]()** контроль границ. Кроме этого, функция **operator[]()** должна возвращать ссылку на индексируемый элемент. Например, в представленном ниже примере в предыдущую программу добавлен контроль границ массива, что позволяет при нарушении границ генерировать соответствующую ошибку.

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

const int SIZE = 5;

class arraytype {
   int a[SIZE];
public:
   arraytype() {
      int i;
      for (i=o; i<SIZE; i++) a[i] = i;
   }
   int &operator[](int i);
};

// Обеспечение контроля границ для массива типа arraytype
int &arraytype::operator[](int i)
{
   if(i<0 || i>SIZE-1) {
      cout << "\nЗначение индекса ";
      cout << i << " находится за пределами границ массива. \n";
      exit(1);
   }
   return a[i];
}

int main()
{
   arraytype ob;
   int i;

   // Здесь проблем нет
   for(i=0; i<SIZE; i++)
      cout << ob[i] << " ";

   /* А здесь при выполнении программы генерируется ошибка, поскольку значение SIZE+100 не входит в
заданный диапазон */
   ob[SIZE+100] = 99; // Ошибка!!!
   return 0;
}
```

Благодаря контролю границ, реализованному в функции **operator[]()**, при выполнении инструкции:

```
ob[SIZE+100] = 99;
```

программа завершится еще до того, как будет повреждена какая-либо ячейка памяти.

Поскольку перегрузка оператора **[]** позволяет создавать безопасные массивы, которые выглядят и функционируют так же, как самые обычные массивы, их можно безболезненно добавить в вашу программную среду. Однако будьте внимательны. Безопасный массив увеличивает расход ресурсов, что не во всех ситуациях может оказаться приемлемым. Фактически, именно из-за непроизводительного расхода ресурсов в C++ отсутствует встроенный контроль границ массивов. Тем не менее, в тех приложениях, в которых желательно обеспечить целостность границ, реализация безопасного массива будет лучшим решением.