# Глава 5
# Перегрузка функций

В этой главе вы более подробно изучите перегрузку функций. Хотя с этой темой вы уже встречались, имеется несколько дополнительных аспектов, с которыми необходимо познакомиться. Здесь вы найдёте ответы на следующие вопросы: как перегрузить конструктор, как создать конструктор копий, как функции передать аргументы по умолчанию, как можно избежать неоднозначности при перегрузке функций.

## 5.1. Перегрузка конструкторов

В программах на C++ перегрузка конструктора класса вполне обычна. (Деструктор, однако, перегружать нельзя.) Имеется три основных причины перегрузки конструктора, которая, как правило, выполняется либо для обеспечения гибкости, либо для поддержки массивов, либо для создания конструкторов копий. В этом разделе рассказывается об обеспечении гибкости и поддержке массивов, а о конструкторах копий — в следующем.

Перед изучением примеров необходимо запомнить одну вещь: каждому способу объявления объекта класса должна соответствовать своя версия конструктора класса. Если эта задача не решена, то при компиляции программы обнаружится ошибка. Именно поэтому перегрузка конструктора столь обычна для программ C++.

### Примеры

1. Вероятно, наиболее частое использование перегрузки конструктора — это обеспечение возможности выбора способа инициализации объекта. Например, в следующей программе объекту **o1** дается начальное значение, а объекту **o2** - нет. Если вы удалите конструктор с пустым списком аргументов, программа не будет компилироваться, поскольку у не инициализируемого объекта типа **samp** не будет конструктора. И наоборот, если вы удалите конструктор с параметром, программа не будет компилироваться, поскольку не будет конструктора у инициализируемого объекта типа **samp.** Для правильной компиляции программы необходимы оба конструктора.

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

class myclass {
   int x;
public:
   // перегрузка конструктора двумя способами
   myclass() { x = 0; } // нет инициализации
   myclass(int n) { x = n; } // инициализация
   int getx() { return x; }
};

int main()
{
   myclass o1(10); // объявление с начальным значением
   myclass o2; // объявление без начального значения

   cout << "o1: " << o1.getx() << '\n';
   cout << "o2: " << o2.getx() << '\n';
```

```
   return 0;
}
```

2. Другой традиционный довод в пользу перегрузки конструктора состоит в том, что такая перегрузка позволяет сосуществовать в программе, как отдельным объектам, так и массивам объектов. Как вы, наверное, знаете по своему опыту программирования, вполне обычно инициализировать отдельную переменную, тогда как инициализация массива встречается достаточно редко. (Гораздо чаще элементам массива присваиваются их значения в зависимости от информации, получаемой уже при выполнении программы.) Таким образом, для сосуществования в программе неинициализированных массивов объектов наряду с инициализированными объектами вы должны использовать конструктор, который поддерживает инициализацию, и конструктор, который ее не поддерживает.

Например, для класса myclass из примера 1 оба этих объявления правильны:

```
myclass ob(10);
myclass ob[5];
```

Обеспечив наличие обоих конструкторов (с параметрами и без параметров), вы в своей программе получаете возможность создавать объекты, которые при необходимости можно либо инициализировать, либо нет.

Естественно, что после определения конструктора с параметрами и конструктора без параметров, их можно использовать для создания инициализированных или неинициализированных массивов. Например, в следующей программе объявляются два массива типа myclass; при этом один из них инициализируется, а другой нет:

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

class myclass {
   int x;
public:
   // перегрузка конструктора двумя способами
   myclass() { x = 0; } // нет инициализации
   myclass(int n) { x = n; } // инициализация
   int getx() { return x; }
};

int main()
{
   myclass o1[10]; // объявление массива без инициализации

   // объявление с инициализацией
   myclass o2[10] = {1,2,3,4,5,6,7,8,9,10};

   int i;

   for(i=0; i<10; i++) {
      cout << "o1[ " << i << "]:" << o1[i].getx();
      cout << '\n';
      cout << "o2[ " << i << "]:" << o2[i].getx();
      cout << '\n';
   }

   return 0;
}
```

В этом примере все элементы массива **o1** конструктор устанавливает в нуль. Элементы массива **o2** инициализируются так, как показано в программе.

3. Другой довод в пользу перегрузки конструкторов состоит в том, что такая перегрузка позволяет программисту выбрать наиболее подходящий метод инициализации объекта. Чтобы понять, как это делается, рассмотрим следующий пример, в котором создается класс для хранения календарной даты. Конструктор **date()** перегружается двумя способами. В первом случае данные задаются в виде строки символов, в другом — в виде трех целых.

```
#include <iostream>
#include <cstdio> // заголовок для функции sscanf()
using namespace std;

class date {
   int day, month, year;
public:
   date(char *str);
   date (int m, int d, int y) {
      day = d;
      month = m;
      year = y;
   }
   void show() {
      cout << month << '/' << day << '/';
      cout << year << '\n';
   }
};

date::date(char *str)
{
   sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);
}

int main()
{
   // использование конструктора для даты в виде строки
   date sdate("11/1/92");

   // использование конструктора для даты в виде трех целых
   date idate(11, 1, 92);

   sdate.show();
   idate.show();

   return 0;
}
```

Преимущество перегрузки конструктора **date(),** как показано в программе, в том, что вы можете выбрать ту версию инициализации, которая лучше всего. подходит к текущей ситуации. Например, если объект типа **date** создается в результате пользовательского ввода, то проще использовать строковую версию. Однако если объект типа **date** строится путем каких-то простых внутренних расчетов, то версия с тремя целыми параметрами становится, вероятно, более привлекательной.

Хотя конструктор можно перегружать любое количество раз, лучше этим не злоупотреблять. С точки зрения стилистики, конструктор имеет смысл перегружать только тогда, когда такая перегрузка позволяет адаптировать программу к часто встречающимся

ситуациям. Например, еще одна перегрузка конструктора **date()** для ввода трех восьмеричных целых вряд ли будет иметь какой-то смысл. Однако перегрузка конструктора **date()** для доступа к объекту типа **time_t** (тип данных для хранения системных даты и времени) могла бы оказаться весьма полезной.

4. Другая ситуация, в которой вам потребуется перегрузить конструктор класса, возникает при выделении динамической памяти массиву объектов этого класса. Как вы должны были узнать из предыдущей главы, динамический массив не может быть инициализирован. Поэтому, если в классе есть инициализирующий конструктор, вам необходимо включить туда и его перегруженную версию без инициализации. Например, ниже приведена программа, в которой массиву объектов динамически выделяется память:

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

class myclass {
   int x;
public:
   // перегрузка конструктора двумя способами
   myclass() { x = 0; } // нет инициализации
   myclass(int n) { x = n; } // инициализация
   int getx() { return x; }
   void setx(int n) { x = n; }
};

int main()
{
   myclass *p;
   myclass ob(10); // инициализация отдельной переменной

   p = new myclass[10]; // здесь инициализировать нельзя
   if(!p) {
      cout << "Ошибка выделения памяти\n";
      return 1;
   }

   int i;

   // инициализация всех элементов значением ob
   for(i=0; i<10; i++) p[i] = ob;

   for(i=0; i<10; i++) {
      cout << "p[ " << i << "]:" << p[i].getx();
      cout << '\n';
   }

   return 0;
}
```

## 5.2. Создание и использование конструкторов копий

Одной из важнейших форм перегруженного конструктора является *конструктор копий (copy constructor).* Как показано в многочисленных примерах из предыдущих глав, передача объектов функциям и их возвращение из функций могут привести к разного рода проблемам. В этом разделе вы узнаете, что одним из способов обойти эти проблемы является определение конструктора копий.

Для начала давайте обозначим проблемы, для решения которых предназначен конструктор копий. Когда объект передается в функцию, делается поразрядная (т. е. точная) копия этого объекта и передается тому параметру Функции, который получает объект. Однако бывают ситуации, в которых такая точная копия объекта нежелательна. Например, если объект содержит Указатель на выделенную область памяти, то в копии указатель будет ссылаться на *ту же самую* область памяти, на которую ссылается исходный Указатель. Следовательно, если копия меняет содержимое области памяти, то эти изменения коснутся также и исходного объекта! Кроме того, когда выполнение функции завершается, копия удаляется, что приводит к вызову Деструктора этой копии. Вызов деструктора может привести к нежелательным побочным эффектам, которые в дальнейшем повлияют на исходный объект.

Сходная ситуация имеет место, когда объект является возвращаемым значением функции. Как правило, компилятор генерирует временный объект для хранения возвращаемого функцией значения. (Это происходит автоматически и незаметно для вас.) Как только значение возвращается и вызывающую процедуру, временный объект выходит из области видимости, что приводит к вызову деструктора временного объекта. Однако если деструктор удаляет что-то необходимое в вызывающей процедуре (например, если он освобождает динамически выделенную область памяти), то это также приводит к проблемам.

В основе этих проблем лежит факт создания поразрядной копии объекта. Для решения задачи вам, как программисту, необходимо предварительно определить всё то, что будет происходить при образовании копии объекта, и таким образом избежать неожиданных побочных эффектов. Способом добиться этого является создание конструктора копий. Путём определения такого конструктора вы можете полностью контролировать весь процесс образования копии объекта.

Важно понимать, что в C++ точно разделяются два типа ситуаций, в которых значение одного объекта передаётся другому. Первая ситуация – это присваивание. Вторая – инициализация, которая может иметь место в трёх случаях:

- Когда в инструкции объявления объекта один объект используется для инициализации другого
- Когда объект передаётся в функцию в качестве параметра
- Когда в качестве возвращаемого значения функции создаётся временный объект

Конструктор копий употребляется только при инициализации, но не для присваивания.

По умолчанию при инициализации компилятор автоматически генерирует код, осуществляющий поразрядное копирование. (То есть C++ автоматически создаёт конструктор копий по умолчанию, который просто дублирует инициализируемый объект.) Однако путём определения конструктора копий вполне возможно предварительно задать то, как один объект будет инициализировать другой. После того как конструктор копий определён, он вызывается всегда при инициализации одного объекта другим.

## Примеры

1. В данном примере показано, почему необходимое явное определение конструктора

копий. В этой программе создаётся простейший «безопасный» массив целых, в котором предотвращёна возможность нарушения границ массива. Память для массива выделяется с помощью оператора **new**, а указатель на эту память поддерживается внутри каждого объекта-массива.

```
/* В этой программе создается класс "безопасный" массив. Поскольку память для массива выделяется динамически, то, когда один массив используется для инициализации другого, для выделения памяти создается конструктор копий.
*/
#include <iostream>
#include <cstdlib>
using namespace std;

class array {
   int *p;
   int size;
public:
   array (int sz) {        // конструктор
   p=new int[sz];
   if(!p) exit(1);
   size=sz;
   cout << "Использование обычного конструктора\n";
   }
   ~ array() {delete [] p;}

   // конструктор копий
   array(const array &a);

   void put(int i, int j) {
      if(i>=0 && i<size) p[i]=j;
   }
   int get(int i) {
      return p[i];
   }
};

/* Конструктор копий.
Память выделяется специально для копии, и адрес этой памяти передается в указатель p. Следовательно, указатель p больше не ссылается на ту же самую, где находится исходный объект, динамически выделенную область памяти:
*/
array:: array(const array &a) {
   int i;

   p=new int[a.size];    // выделение памяти для копии
   if(!p) exit(1);
   for(i=0; i<a.size; i++) p[i]=a.p[i]; // копирование содержимого
   cout << "Использование конструктора копий\n";
}

int main()
```

```
{
  array num(10);       // вызов обычного конструктора
  int i;

  // помещение в массив нескольких значений
  for(i=0; i<10; i++) num.put(i, i);

  // вывод на экран массива num
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "\n";

  // создание другого массива и инициализация его массивом num
  array x=num;       // вызов конструктора копий

  // вывод на экран массива x
  for(i=0; i<10; i++) cout << x.get(i);

  return 0;
}
```

Когда массив **num** используется для инициализации массива **x**, вызывается конструктор копий и для нового массива по адресу **x.p** выделяется память, а содержимое массива **num** копируется в массив **x**. В этом случае в массивах **x** и **num** находятся одинаковые значения, но при этом – это совершенно различные массивы. (Другими словами, указатели **x.p** и **num.p** теперь не ссылаются на одну и ту же область памяти.) Если бы не был создан конструктор копий, то поразрядная инициализация при выполнении инструкции **array x=num** привела бы к тому, что массивы **x** и **num** оказались бы в одной и той же области памяти! (То есть указатели **x.p** и **num.p** ссылались бы на одну и ту же область памяти.)

Конструктор копий вызывается только при инициализации. Например, следующая последовательность инструкций не ведёт к вызову определённого в предыдущей программе конструктора копий:

```
array a(10);
array b(10);

b=a; //конструктор копий не вызывается
```

В данном случае инструкция **b=a** представляет собой операцию присваивания.

2. Чтобы понять, как конструктор копий помогает предотвратить некоторые проблемы, связанные с передачей функциям объектов определённых типов, рассмотрим следующую, неправильную программу:

```
// В этой программе имеется ошибка
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
  char *p;
public:
  strtype(char *s);
```

```
   ~strtype() {delete [] p;}
   char *get() {return p;}
};

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

   l=strlen(s)+1;

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

   strcpy(p, s);
}

void show(strtype x)
{
   char *s;

   s=x.get();
   cout << s << "\n";
}

int main()
{
strtype a("Hello"), b("There");

show(a);
show(b);

return 0;
}
```

В этой программе, когда объект типа **strtype** передаётся в функцию **show()**, создаётся поразрядная копия объекта (поскольку не был определён конструктор копий) и передаётся параметру **x**. Таким образом, когда функция возвращает своё значение, **x** выходит из области видимости и удаляется. Это, естественно, приводит к вызову деструктора объекта **x**, который освобождает область памяти по адресу **x.p**. Однако освобождённая память – это та самая память, которую продолжает занимать объект, используемый при вызове функции. Это приводит к ошибке.

Решение предыдущей проблемы лежит в определении конструктора копий для класса **strtype**, который при создании копии объекта типа **strtype** выделяет для неё память. Такой подход используется в следующей, исправленной версии программы:

```
/* В этой программе используется конструктор копирования, что позволяет передавать функции объекты типа
strtype. */
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype {
   char *p;
public:
   strtype(char *s);        // конструктор
```

```
   strtype(const strtype &o);    // конструктор копий
   ~strtype() {delete [] p;}        // деструктор
   char *get() {return p;}
};

// Обычный конструктор
strtype::strtype(char *s)
{
   int l;

   l=strlen(s)+1;

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

   strcpy(p, s);
}

// Конструктор копий
strtype::strtype(const strtype &o)
{
   int l;

   l=strlen(o.p)+1;

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

   strcpy(p, o.p);        // копирование строки в копию
}

void show(strtype x)
{
   char *s;

   s=x.get();
   cout << s << "\n";
}

int main()
{
strtype a("Hello"), b("There");

show(a);
show(b);

return 0;
}
```

Теперь, когда функция **show()** завершается и объект **x** выходит из области видимости, память, на которую ссылается указатель **x.p** (освобождаемая память), - это уже не та память, которая используется переданным в функцию объектом.

## 5.3. Устаревшее ключевое слово ***overload.***

В ранних версиях C++ для создания перегружаемых функций требовалось ключевое слово **overload.** Хотя ключевое слово **overload в** современных компиляторах больше не поддерживается, вы все еще можете встретить его в существующих программах, поэтому полезно знать, как оно использовалось.

Ниже показана основная форма ключевого слова **overload:**

**overload** ***имя_функции;***

Здесь ***имя_функции*** — это имя перегружаемой функции. Перед этой инструкцией должно находиться объявление перегружаемой функции. Например, следующая инструкция сообщает компилятору, что вы будете перегружать функцию **timer():**

```
overload timer ();
```

## 5.4. Аргументы по умолчанию.

В синтаксисе C++ имеется элемент, который имеет непосредственное отношение к перегрузке функций и называется *аргументом по умолчанию (default argument).* Аргумент по умолчанию позволяет вам, если при вызове функции соответствующий аргумент не задан, присвоить параметру значение по умолчанию. Как вы увидите далее, применение аргумента по умолчанию является скрытой формой перегрузки функций.

Чтобы передать параметру аргумент по умолчанию, нужно в инструкции определения функции приравнять параметр тому значению, которое вы хотите передать, когда при вызове функции соответствующий аргумент не будет указан. Например, в представленной ниже функции двум параметрам по умолчанию присваивается значение 0:

```
void f(int a = 0, int b = 0);
```

Обратите внимание, что данный синтаксис напоминает инициализацию переменных. Теперь эту функцию можно вызвать тремя различными способами. Во-первых, она может вызываться с двумя заданными аргументами. Во-вторых, она может вызываться только с первым заданным аргументом. В этом случае параметр **b** по умолчанию станет равным нулю. Наконец, функция **f()** может вызываться вообще без аргументов, при этом параметры **a** и **b** по умолчанию станут равными нулю. Таким образом, все следующие вызовы функции **f()** правильны:

```
f();            // a и b до умолчанию равны 0
f(10);          // a равно 10, b по умолчанию равно 0
f(10, 99);      // a равно 10, b равно 99
```

Из этого примера должно быть ясно, что невозможно передать по умолчанию значение **a** и при этом задать **b**.

Когда вы создаете функцию, имеющую один или более передаваемых по умолчанию аргументов, эти аргументы должны задаваться только один раз: либо в прототипе функции, либо в ее определении, если определение предшествует первому использованию функции. Аргументы по умолчанию нельзя задавать одновременно в определении и в прототипе функции. Это правило остается в силе, даже если вы просто дублируете одни и те же аргу-

менты по умолчанию.

Как вы, вероятно, догадываетесь, все параметры, задаваемые по умолчанию, должны указываться правее параметров, передаваемых обычным путем. Больше того, после того как вы начали определять параметры по умолчанию, параметры, которые по умолчанию не передаются, уже определять
нельзя.

Еще несколько слов об аргументах по умолчанию: они должны быть константами или глобальными переменными. Они не могут быть локальными переменными или другими параметрами.

## Примеры

1. Программа для иллюстрации вышеописанной функции:

```
// Первый простой пример аргументов по умолчанию
#include <iostream>
using namespace std;

void f(int a = 0, int b = 0)
{
   cout << "a: " << a << ", b: " << b;
   cout << '\n';
}

int main()
{
   f();
   f(10);
   f(10, 99);

   return 0;
}
```

Как и следовало ожидать, на экран выводится следующее:

```
a:     0,   b: 0
a:     10, b: 0
a:     10, b: 99
```

Запомните, если первый аргумент задан по умолчанию, все последующие параметры должны также задаваться по умолчанию. Например, такое небольшое изменение функции **f()** приведет к ошибке при компиляции программы:

```
void f(int a = 0, int b) // Неправильно! Параметр b тоже должен
        // задаваться по умолчанию
{
cout << "a: " << a << ", b: " << b;
cout << '\n' ;
}
```

2. Чтобы понять, какое отношение аргументы по умолчанию имеют к перегрузке функций, рассмотрим следующую программу, в которой перегружается функция **rect_area().** Эта функция возвращает площадь прямоугольника.

```
// Расчет площади прямоугольника с использованием перегрузки функций
#include <iostream>
using namespace std;

// Возвращает площадь неравностороннего прямоугольника
double rect_area(double length, double width)
{
   return length * width;
}

// Возвращает площадь квадрата
double rect_area(double length)
{
   return length * length;
}

int main()
{
   cout << "площадь прямоугольника 10 x 5.8 равна: ";
   cout << rect_area(10.0, 5.8) << '\n';

   cout << "площадь квадрата 10 x 10 равна: ";
   cout << rect_area(10.0) << '\n';

   return 0;
}
```

В этой программе функция **rect_area()** перегружается двумя способами. В первом — функции передаются оба размера фигуры. Эта версия используется для прямоугольника. Однако в случае квадрата необходимо задавать только один аргумент, поэтому вызывается вторая версия функции **rect_area().**

Если исследовать этот пример, то становится ясно, что на самом деле в такой ситуации нет необходимости в двух функциях. Вместо этого второму параметру можно по умолчанию передать некоторое значение, действующее как флаг для функции **rect_area().** Когда функция встретит это значение, она дважды использует параметр **length.** Пример такого подхода:

```
   // Расчет площади прямоугольника с передачей аргументов по умолчанию
   #include <iostream>
   using namespace std;

   // Возвращает площадь прямоугольника
   double rect_area(double length, double width = 0)
   {
      if(!width) width = length;
      return length * width;
   }

   int main()
   {
      cout << "площадь прямоугольника 10 x 5.8 равна: ";
      cout << rect_area(10.0, 5.8) << '\n';

      cout << "площадь квадрата 10 x 10 равна: ";
      cout << rect_area(10.0) << '\n';

      return 0;
}
```

Теперь параметру **width** по умолчанию присваивается нуль. Такое значение выбрано потому, что не бывает прямоугольника с нулевой стороной. (Фактически, прямоугольник с нулевой стороной — это линия.) Таким образом, когда в **rect_area()** встречается такое, переданное по умолчанию значение, для ширины прямоугольника автоматически используется параметр **length.**

Как показано в этом примере, аргументы по умолчанию часто обеспечивают простую альтернативу перегрузке функций. (Конечно, имеется масса ситуаций. в которых перегрузка функций необходима по-прежнему.)

3. Передавать конструкторам аргументы по умолчанию не только правильно, но и вполне обычно. Как отмечалось ранее в этой главе, часто конструктор перегружается просто для того, чтобы могли создаваться как инициализируемые, так и неинициализируемые объекты- Во многих случаях можно Избежать перегрузки конструктора путем передачи ему одного или более аргументов по умолчанию. Например, рассмотрим следующую программу:

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

class myclass {
   int x;
public:
/* Использование аргумента по умолчанию вместо перегрузки конструктора
*/
   myclass(int n = 0) { x = n; }
   int getx() { return x; }
};

int main()
{
   myclass o1(10); // объявление с начальным значением
   myclass o2; // объявление без начального значения

   cout << "o1: " << o1.getx() << '\n';
   cout << "o2: " << o2.getx() << '\n';

   return 0;
}
```

Как показано в этом примере, путем передачи по умолчанию параметру **n** нулевого значения, можно создавать не только объекты, имеющие явно заданные начальные значения, но и такие, для которых достаточно значений,
задаваемых по умолчанию.

4. Другим хорошим применением аргумента по умолчанию является случай, когда с помощью такого параметра происходит выбор нужного варианта развития событий. Можно передать параметру значение по умолчанию так, чтобы использовать его в качестве флага, сообщающего функции о необходимости продолжить работу в обычном режиме- Например, в следующей программе функция **print()** выводит строку на экран- Если параметр **how** равен значению **ignore,** текст выводится в том виде, в каком он задан. Если параметр **how** равен значению **upper,** текст выводится в верхнем регистре. Если параметр **how** равен значению **lower,** текст выводится в нижнем регистре. Если параметр **how** не задан, его значение по умолчанию равно —1, что говорит функции о необходимости повторно использовать его предыдущее значение.

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

const int ignore = 0;
const int upper = 1;
const int lower = 2;

void print(char *s, int how = -1);

int main()
{
   print("Привет \n", ignore);
   print("Привет \n", upper);
   print("Привет \n"); // продолжение вывода в верхнем регистре
   print("Привет \n", lower);
   print("Это конец \n"); // продолжение вывода в нижнем регистре

   return 0;
}

/* Печать строки в заданном регистре. Использование заданного последним регистра, если он не задан.
*/
void print(char *s, int how)
{
   static int oldcase = ignore;

   // повторять работу с прежним регистром, если новый не задан
   if(how<0) how = oldcase;
   while (*s) {
      switch(how) {
         case upper: cout << (char) toupper(*s);
            break;
         case lower: cout << (char) tolower(*s);
            break;
         default: cout << *s;
      }
      s++;
   }
   oldcase = how;
}
```

Эта программа выводит следующее:

```
Привет
ПРИВЕТ
ПРИВЕТ
привет
это конец
```

5. Ранее в этой главе мы рассматривали общую форму конструктора копий. В этой общей форме имелся только один параметр. Однако вполне возможно создавать конструкторы копий, получающие дополнительные аргументы, если только это аргументы по умолчанию. Например, вполне приемлема следующая форма конструктора копий:

```
myclass(const myclass &obj, int x=0) {
        // тело конструктора
}
```

Поскольку первый аргумент является ссылкой на копируемый объект, а все остальные — это аргументы по умолчанию, эту функцию можно квалифицировать как конструктор копий. Такая гибкость позволяет создавать самые разнообразные конструкторы копий.

6. Хотя аргументы по умолчанию являются мощным и удобным инструментом, ими нельзя злоупотреблять. Несомненно, что при правильном применении аргументы по умолчанию позволяют функции выполнять свою работу эффективным и простым по реализации образом. Однако так происходит лишь тогда, когда переданное по умолчанию значение имеет смысл. Например, если аргумент, используемый в девять или десять раз чаще других, передать функции по умолчанию, то, очевидно, это неплохо. Однако в случае, если нет значения, используемого чаще других, или нет выгоды от аргумента по умолчанию в качестве флага, то нет большого смысла передавать что-либо по умолчанию. Фактически, обеспечение передачи аргумента по умолчанию, когда это не вызвано необходимостью, ограничивает возможности вашей программы и вводит в заблуждение всех пользователей такой функции.

   Как и при перегрузке функций, хороший программист в каждом конкретном случае всегда сумеет определить, стоит или нет пользоваться аргументом по умолчанию.

## 5.5. Перегрузка и неоднозначность.

При перегрузке возможно внесение неоднозначности в программу. *Неоднозначность (ambiguity),* вызванная перегрузкой функций, может быть введена в программу при преобразовании типа, а также при использовании параметров-ссылок и аргументов по умолчанию. Некоторые виды неоднозначности вызываются самой перегрузкой функций. Другие виды связаны со способом вызова перегруженных функций. Чтобы программа компилировалась без ошибок, от неоднозначности необходимо избавиться.

## Примеры

1. Один из наиболее частых видов неоднозначности вызывается правилами преобразования типа в C++. Как вы знаете, при вызове функции с аргументом, тип которого совместим (но не аналогичен) с типом параметра, которому он передается, тип аргумента по умолчанию преобразуется в тип параметра. Об этой операции иногда говорят как о *приведении типа (type promotion).* Приведение типа — это такой вид преобразования типа, который позволяет некоторым функциям, например **putchar(),** вызываться с символьным параметром, даже тогда, когда аргумент функции имеет тип **int.** Однако в некоторых случаях это преобразование типа при перегрузке функций вызовет ситуацию неоднозначности. Чтобы понять, как это происходит, исследуем следующую программу:

```
// Эта программа содержит ошибку неоднозначности.
#include <iostream>
using namespace std;

float f(float i)
{
  return i / 2.0;
}

double f(double i)
{
```

```
   return i / 3.0;
}

int main()
{
   float x = 10.09;
   double y = 10.09;

   cout << f(x); // нет неоднозначности
                 // используется функция f(float)
   cout << f(y); // нет неоднозначности
                 // используется функция f(double)

   cout << f(10); // неоднозначность
                  // куда преобразовать 10?
                  // в значение типа double или float?

   return 0;
}
```

Как указано в комментариях к функции **main(),** компилятор в состоянии выбрать правильную версию функции f(), если она вызывается либо с переменными типа **double,** либо с переменными типа **float.** Однако что случается, если она вызывается с целым? Какую функцию вызовет компилятор f(float) или **f(double)?** (Оба преобразования правильны!) И в том, и в другом случае правильно "привести" тип **int** либо к типу **float,** либо к типу **double.** Таким образом, возникает ситуация неоднозначности.

Этот пример выявляет также то, как неоднозначность может проявляться при вызове перегруженных функций. Очевидно, что сама по себе неоднозначность не присуща перегруженным версиям функции **f(),** пока каждая вызывается с аргументом соответствующего типа.

2. Другой пример перегрузки функции, которая сама по себе не должна приводить к неоднозначности. Тем не менее, при вызове с аргументом неправильного типа, правила преобразования типа C++ создают ситуацию неоднозначности.

```
// Эта программа неоднозначна
#include <iostream>
using namespace std;

void f(unsigned char c)
{
   cout << c;
}

void f(char c)
{
   cout << c;
}

int main()
{
   f('c');
   f(86); // какая версия f() вызывается?

   return 0;
}
```

Когда функция **f()** вызывается с числовой константой 86, компилятор не может понять, какую версию функции вызвать: **f(unsigned char)** или **f(char).** Оба преобразования одинаково правильны, что и ведет к неоднозначности.

3. Один из видов неоднозначности проявляется, если вы пытаетесь перегрузить функции, единственным отличием которых является то, что одна использует параметр-ссылку, а другая параметр-значение по умолчанию. В рамках формального синтаксиса C++ у компилятора нет способа узнать, какую функцию вызвать. Запомните, что нет синтаксических отличий между вызовом функции по значению и вызовом функции по ссылке. Например:

```
// Эта программа неоднозначна
#include <iostream>
using namespace std;

int f(int a, int b)
{
   return a + b;
}

// здесь внутренняя неоднозначность
int f(int a, int &b)
{
   return a - b;
}

int main()
{
   int x = 1, y = 2;

   cout << f(x, y); // какую версию f() вызвать?

   return 0;
}
```

Здесь вызов функции **f(x, y)** неоднозначен, поскольку вызвана может быть любая версия функции. При этом компилятор выставит флаг ошибки даже раньше того, как встретится такая инструкция, поскольку сама перегрузка этих двух функций внутренне неоднозначна, и компилятор не будет знать, какую из них предпочесть.

4. Другим видом неоднозначности при перегрузке функций является случай, Когда одна или более перегруженных функций используют аргумент по Умолчанию. Рассмотрим программу:

```
// Неоднозначность, основанная на аргументах по умолчанию
// и перегрузке функций
#include <iostream>
using namespace std;

int f(int a)
{
   return a * a;
}

int f(int a, int b = 0)
{
   return a * b;
}
```

```
int main()
{
   cout << f(10, 2); // вызывается f(int, int)
   cout << f(10); // неоднозначность,
                  // что вызвать f(int, int) или f(int)???

   return 0;
}
```

Здесь вызов функции **f(10,** 2) совершенно правилен и не ведет к неоднозначности. Однако у компилятора нет способа выяснить, какую версию функции **f()** вызывает версия **f(10)** — первую или вторую, в которой параметр **b** передается по умолчанию.

## 5.5. Определение адреса перегруженной функции.

В заключение этой главы вы узнаете, как найти адрес перегруженной функции. Так же, как и в С, вы можете присвоить адрес функции указателю и получить доступ к функции через этот указатель. Адрес функции можно найти, если поместить имя функции в правой части инструкции присваивания без всяких скобок или аргументов. Например, если **zap()** — это функция, причем правильно объявленная, то корректным способом присвоить переменной **p** адрес функции **zap()** является инструкция:

```
p = zap;
```

В языке С любой тип указателя может использоваться как указатель на функцию, поскольку имеется только одна функция, на которую он может ссылаться. Однако в С++ ситуация несколько более сложная, поскольку функция может быть перегружена. Таким образом, должен быть некий механизм, который позволял **бы** определять адреса перегруженных версий функции.

решение оказывается не только элегантным, но и эффектным. *Способ объявления указателя* и определяет то, адрес какой из перегруженных версий функции будет получен. Уточним, объявления указателей соответствуют объявлениям перегруженных функций. Функция, объявлению которой соответствует объявление указателя, и является искомой функцией.

## Примеры

1. Здесь представлена программа, которая содержит две версии функции **space().** Первая версия выводит на экран некоторое число пробелов, заданное в переменной **count.** Вторая версия выводит на экран некоторое число каких-то иных символов, вид которых задан в переменной **ch.** В функции **mainO** объявляются оба указателя на эти функции. Первый задается как указатель на функцию, имеющую только один целый параметр. Второй объявляется как указатель на функцию, имеющую два параметра.

```
/* Иллюстрация присваивания и получения указателей на перегруженные функции.
*/
#include <iostream>
using namespace std;

// вывод заданного в переменной count числа пробелов
void space(int count)
{
   for( ; count; count --) cout << ' ';
```

```
}

// вывод заданного в переменной count числа символов,
// вид которых задан в переменной ch
void space(int count, char ch)
{
   for( ; count; count --) cout << ch;
}

int main()
{
   /* Создание указателя на функцию с одним целым параметром. */
   void (*fp1)(int);

   /* Создание указателя на функцию с одним целым и одним
      символьным параметром. */
   void (*fp2)(int, char);

   fp1 = space; // получение адреса функции space(int)

   fp2 = space; // получение адреса функции space(int, char)

   fp1(22); // выводит 22 пробела
   cout << "|\n";

   fp2(30, 'x'); // выводит 30 символов x
   cout << "|\n";

   return 0;
}
```

**Как** показано в комментариях, на основе того, **каким** образом объявляются указатели **fpl и fp2,** компилятор способен определить, какой из них на какую из перегруженных функций будет ссылаться.

Повторим, если вы присваиваете адрес перегруженной функции указателю на функцию, то объявление указателя определяет, адрес какой именно функции ему присваивается. Более того, объявление указателя на функцию должно точно соответствовать одной и только одной перегруженной функции. Если это не так, будет внесена неоднозначность, что приведет к ошибке при компиляции программы.