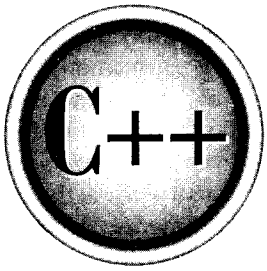
The Complete Reference



Chapter 14

Function Overloading, Copy Constructors, and Default Arguments

This chapter examines function overloading, copy constructors, and default arguments. Function overloading is one of the defining aspects of the C++ programming language. Not only does it provide support for compile-time polymorphism, it also adds flexibility and convenience. Some of the most commonly overloaded functions are constructors. Perhaps the most important form of an overloaded constructor is the copy constructor. Closely related to function overloading are default arguments. Default arguments can sometimes provide an alternative to function overloading.

Function Overloading

Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation. For example, this program overloads **myfunc( )** by using different types of parameters.

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

int myfunc(int i); // these differ in types of parameters
double myfunc(double i);

int main()
{

  cout << myfunc(10) << " "; // calls myfunc(int i)
  cout << myfunc(5.4); // calls myfunc(double i)

  return 0;
}

double myfunc(double i)
{
  return i;
}

int myfunc(int i)
{
  return i;
}
```

The next program overloads **myfunc( )** using a different number of parameters:

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

int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);

int main()
{
  cout << myfunc(10) << " "; // calls myfunc(int i)
  cout << myfunc(4, 5); // calls myfunc(int i, int j)

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

As mentioned, the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded. For example, this is an invalid attempt to overload **myfunc( )**:

```
int myfunc(int i);   // Error: differing return types are
float myfunc(int i); // insufficient when overloading.
```

Sometimes, two function declarations will appear to differ, when in fact they do not. For example, consider the following declarations.

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

Remember, to the compiler ***p** is the same as **p[ ]**. Therefore, although the two prototypes appear to differ in the types of their parameter, in actuality they do not.

Overloading Constructors

Constructors can be overloaded; in fact, overloaded constructors are very common. There are three main reasons why you will want to overload a constructor: to gain flexibility, to allow both initialized and uninitialized objects to be created, and to define copy constructors. In this section, the first two of these are examined. The following section describes the copy constructor.

Overloading a Constructor to Gain Flexibility

Many times you will create a class for which there are two or more possible ways to construct an object. In these cases, you will want to provide an overloaded constructor for each way. This is a self-enforcing rule because if you attempt to create an object for which there is no matching constructor, a compile-time error results.

By providing a constructor for each way that a user of your class may plausibly want to construct an object, you increase the flexibility of your class. The user is free to choose the best way to construct an object given the specific circumstance. Consider this program that creates a class called **date**, which holds a calendar date. Notice that the constructor is overloaded two ways:

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

class date {
  int day, month, year;
public:
  date(char *d);
  date(int m, int d, int y);
  void show_date();
};

// Initialize using string.
date::date(char *d)
{
  sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);
}

// Initialize using integers.
date::date(int m, int d, int y)
{
```

```
  day = d;
  month = m;
  year = y;
}

void date::show_date()
{
  cout << month << "/" << day;
  cout << "/" << year << "\n";
}

int main()
{
  date ob1(12, 4, 2003), ob2("10/22/2003");

  ob1.show_date();
  ob2.show_date();

  return 0;
}
```

In this program, you can initialize an object of type **date**, either by specifying the date using three integers to represent the month, day, and year, or by using a string that contains the date in this general form:

mm/dd/yyyy

Since both are common ways to represent a date, it makes sense that **date** allow both when constructing an object.

As the **date** class illustrates, perhaps the most common reason to overload a constructor is to allow an object to be created by using the most appropriate and natural means for each particular circumstance. For example, in the following **main( )**, the user is prompted for the date, which is input to array **s**. This string can then be used directly to create **d**. There is no need for it to be converted to any other form. However, if **date( )** were not overloaded to accept the string form, you would have to manually convert it into three integers.

```
int main()
{
  char s[80];
```

```
  cout << "Enter new date: ";
  cin >> s;

  date d(s);
  d.show_date();

  return 0;
}
```

In another situation, initializing an object of type **date** by using three integers may be more convenient. For example, if the date is generated by some sort of computational method, then creating a **date** object using **date(int, int, int)** is the most natural and appropriate constructor to employ. The point here is that by overloading **date**'s constructor, you have made it more flexible and easier to use. This increased flexibility and ease of use are especially important if you are creating class libraries that will be used by other programmers.

Allowing Both Initialized and Uninitialized Objects

Another common reason constructors are overloaded is to allow both initialized and uninitialized objects (or, more precisely, default initialized objects) to be created. This is especially important if you want to be able to create dynamic arrays of objects of some class, since it is not possible to initialize a dynamically allocated array. To allow uninitialized arrays of objects along with initialized objects, you must include a constructor that supports initialization and one that does not.

For example, the following program declares two arrays of type **powers**; one is initialized and the other is not. It also dynamically allocates an array.

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

class powers {
  int x;
public:
  // overload constructor two ways
  powers() { x = 0; } // no initializer
  powers(int n) { x = n; } // initializer

  int getx() { return x; }
  void setx(int i) { x = i; }
```

```
};

int main()
{
  powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
  powers ofThree[5]; // uninitialized
  powers *p;
  int i;

  // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
    cout << ofTwo[i].getx() << " ";
  }
  cout << "\n\n";

  // set powers of three
  ofThree[0].setx(1);
  ofThree[1].setx(3);
  ofThree[2].setx(9);
  ofThree[3].setx(27);
  ofThree[4].setx(81);

  // show powers of three
  cout << "Powers of three: ";
  for(i=0; i<5; i++) {
    cout << ofThree[i].getx() << " ";
  }
  cout << "\n\n";

  // dynamically allocate an array
  try {
    p = new powers[5]; // no initialization
  } catch (bad_alloc xa) {
      cout << "Allocation Failure\n";
      return 1;
  }

  // initialize dynamic array with powers of two
  for(i=0; i<5; i++) {
    p[i].setx(ofTwo[i].getx());
```

```
  }

  // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
    cout << p[i].getx() << " ";
  }
  cout << "\n\n";

  delete [] p;
  return 0;
}
```

In this example, both constructors are necessary. The default constructor is used to construct the uninitialized **ofThree** array and the dynamically allocated array. The parameterized constructor is called to create the objects for the **ofTwo** array.

Copy Constructors

One of the more important forms of an overloaded constructor is the *copy constructor*. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another.

Let's begin by restating the problem that the copy constructor is designed to solve. By default, when one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object. Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created. For example, assume a class called *MyClass* that allocates memory for each object when it is created, and an object *A* of that class. This means that *A* has already allocated its memory. Further, assume that *A* is used to initialize *B*, as shown here:

```
MyClass B = A;
```

If a bitwise copy is performed, then *B* will be an exact copy of *A*. This means that *B* will be using the same piece of allocated memory that *A* is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if *MyClass* includes a destructor that frees the memory, then the same piece of memory will be freed twice when *A* and *B* are destroyed!

The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary

objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances.

To solve the type of problem just described, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. Thus, your copy constructor bypasses the default bitwise copy. The most common general form of a copy constructor is

```
classname (const classname &o) {
  // body of constructor
}
```

Here, *o* is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them. However, in all cases the first parameter must be a reference to the object doing the initializing.

It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first is assignment. The second is initialization, which can occur any of three ways:

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, each of the following statements involves initialization.

```
myclass x = y; // y explicitly initializing x
func(y);       // y passed as a parameter
y = func();    // y receiving a temporary, return object
```

Following is an example where an explicit copy constructor is needed. This program creates a very limited "safe" integer array type that prevents array boundaries from being overrun. (Chapter 15 shows a better way to create a safe array that uses overloaded operators.) Storage for each array is allocated by the use of **new**, and a pointer to the memory is maintained within each array object.

```
/* This program creates a "safe" array class.  Since space
   for the array is allocated using new, a copy constructor
   is provided to allocate memory when one array object is
   used to initialize another.
*/
```

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

class array {
  int *p;
  int size;
public:
  array(int sz) {
    try {
      p = new int[sz];
    } catch (bad_alloc xa) {
      cout << "Allocation Failure\n";
      exit(EXIT_FAILURE);
    }
    size = sz;
  }
  ~array() { delete [] p; }

  // copy constructor
  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];
  }
};

// Copy Constructor
array::array(const array &a) {
  int i;

  try {
    p = new int[a.size];
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    exit(EXIT_FAILURE);
  }
  for(i=0; i<a.size; i++) p[i] = a.p[i];
}
```

```
int main()
{
  array num(10);
  int i;

  for(i=0; i<10; i++) num.put(i, i);
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "\n";

  // create another array and initialize with num
  array x(num); // invokes copy constructor
  for(i=0; i<10; i++) cout << x.get(i);

  return 0;
}
```

Let's look closely at what happens when **num** is used to initialize **x** in the statement

```
array x(num); // invokes copy constructor
```

The copy constructor is called, memory for the new array is allocated and stored in **x.p**, and the contents of **num** are copied to **x**'s array. In this way, **x** and **num** have arrays that contain the same values, but each array is separate and distinct. (That is, **num.p** and **x.p** do not point to the same piece of memory.) If the copy constructor had not been created, the default bitwise initialization would have resulted in **x** and **num** sharing the same memory for their arrays. (That is, **num.p** and **x.p** would have indeed pointed to the same location.)

Remember that the copy constructor is called only for initializations. For example, this sequence does not call the copy constructor defined in the preceding program:

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

b = a; // does not call copy constructor
```

In this case, **b = a** performs the assignment operation. If = is not overloaded (as it is not here), a bitwise copy will be made. Therefore, in some cases, you may need to overload the = operator as well as create a copy constructor to avoid certain types of problems (see Chapter 15).

Finding the Address of an Overloaded Function

As explained in Chapter 5, you can obtain the address of a function. One reason to do so is to assign the address of the function to a pointer and then call that function through that pointer. If the function is not overloaded, this process is straightforward. However, for overloaded functions, the process requires a little more subtlety. To understand why, first consider this statement, which assigns the address of some function called **myfunc( )** to a pointer called **p**:

```
p = myfunc;
```

If **myfunc( )** is not overloaded, there is one and only one function called **myfunc( )**, and the compiler has no difficulty assigning its address to **p**. However, if **myfunc( )** is overloaded, how does the compiler know which version's address to assign to **p**? The answer is that it depends upon how **p** is declared. For example, consider this program:

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

int myfunc(int a);
int myfunc(int a, int b);

int main()
{
  int (*fp)(int a); // pointer to int f(int)

  fp = myfunc; // points to myfunc(int)

  cout << fp(5);

  return 0;
}

int myfunc(int a)
{
  return a;
}

int myfunc(int a, int b)
{
  return a*b;
}
```

Here, there are two versions of **myfunc( )**. Both return **int**, but one takes a single integer argument; the other requires two integer arguments. In the program, **fp** is declared as a pointer to a function that returns an integer and that takes one integer argument. When **fp** is assigned the address of **myfunc( )**, C++ uses this information to select the **myfunc(int a)** version of **myfunc( )**. Had **fp** been declared like this:

```
int (*fp)(int a, int b);
```

then **fp** would have been assigned the address of the **myfunc(int a, int b)** version of **myfunc( )**.

In general, when you assign the address of an overloaded function to a function pointer, it is the declaration of the pointer that determines which function's address is obtained. Further, the declaration of the function pointer must exactly match one and only one of the overloaded function's declarations.

The overload Anachronism

When C++ was created, the keyword **overload** was required to create an overloaded function. It is obsolete and no longer used or supported. Indeed, it is not even a reserved word in Standard C++. However, because you might encounter older programs, and for its historical interest, it is a good idea to know how **overload** was used. Here is its general form:

overload *func-name;*

Here, *func-name* is the name of the function that you will be overloading. This statement must precede the overloaded declarations. For example, this tells an old-style compiler that you will be overloading a function called **test( )**:

```
overload test;
```

Default Function Arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization. For example, this declares **myfunc( )** as taking one **double** argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
  // ...
}
```

Now, **myfunc( )** can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
myfunc();        // let function use default
```

The first call passes the value 198.234 to **d**. The second call automatically gives **d** the default value zero.

One reason that default arguments are included in C++ is because they provide another method for the programmer to manage greater complexity. To handle the widest variety of situations, quite frequently a function contains more parameters than are required for its most common usage. Thus, when the default arguments apply, you need specify only the arguments that are meaningful to the exact situation, not all those needed by the most general case. For example, many of the C++ I/O functions make use of default arguments for just this reason.

A simple illustration of how useful a default function argument can be is shown by the **clrscr( )** function in the following program. The **clrscr( )** function clears the screen by outputting a series of linefeeds (not the most efficient way, but sufficient for this example). Because a very common video mode displays 25 lines of text, the default argument of 25 is provided. However, because some video modes display more or less than 25 lines, you can override the default argument by specifying one explicitly.

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

void clrscr(int size=25);

int main()
{
  register int i;

  for(i=0; i<30; i++) cout << i << endl;
  cin.get();
  clrscr(); // clears 25 lines

  for(i=0; i<30; i++) cout << i << endl;
  cin.get();
  clrscr(10); // clears 10 lines

  return 0;
}

void clrscr(int size)
```

```
{
  for(; size; size--) cout << endl;
}
```

As this program illustrates, when the default value is appropriate to the situation, no argument need be specified when **clrscr( )** is called. However, it is still possible to override the default and give **size** a different value when needed.

A default argument can also be used as a flag telling the function to reuse a previous argument. To illustrate this usage, a function called **iputs( )** is developed here that automatically indents a string by a specified amount. To begin, here is a version of this function that does not use a default argument:

```
void iputs(char *str, int indent)
{
  if(indent < 0) indent = 0;

  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

This version of **iputs( )** is called with the string to output as the first argument and the amount to indent as the second. Although there is nothing wrong with writing **iputs( )** this way, you can improve its usability by providing a default argument for the **indent** parameter that tells **iputs( )** to indent to the previously specified level. It is quite common to display a block of text with each line indented the same amount. In this situation, instead of having to supply the same **indent** argument over and over, you can give **indent** a default value that tells **iputs( )** to indent to the level of the previous call. This approach is illustrated in the following program:

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

/* Default indent to -1.  This value tells the function
   to reuse the previous value. */
void iputs(char *str, int indent = -1);

int main()
{
  iputs("Hello there", 10);
  iputs("This will be indented 10 spaces by default");
```

```
  iputs("This will be indented 5 spaces", 5);
  iputs("This is not indented", 0);

  return 0;
}

void iputs(char *str, int indent)
{
  static i = 0; // holds previous indent value

  if(indent >= 0)
    i = indent;
  else  // reuse old indent value
    indent = i;

  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

This program displays this output:

```
          Hello there
          This will be indented 10 spaces by default
     This will be indented 5 spaces
This is not indented
```

When you are creating functions that have default arguments, it is important to remember that the default values must be specified only once, and this must be the first time the function is declared within the file. In the preceding example, the default argument was specified in **iputs( )**'s prototype. If you try to specify new (or even the same) default values in **iputs( )**'s definition, the compiler will display an error and not compile your program. Even though default arguments for the same function cannot be redefined, you can specify different default arguments for each version of an overloaded function.

All parameters that take default values must appear to the right of those that do not. For example, it is incorrect to define **iputs( )** like this:

```
// wrong!
void iputs(int indent = -1, char *str);
```

Once you begin to define parameters that take default values, you cannot specify a nondefaulting parameter. That is, a declaration like this is also wrong and will not compile:

```
int myfunc(float f, char *str, int i=10, int j);
```

Because **i** has been given a default value, **j** must be given one too.

You can also use default parameters in an object's constructor. For example, the **cube** class shown here maintains the integer dimensions of a cube. Its constructor defaults all dimensions to zero if no other arguments are supplied, as shown here:

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

class cube {
  int x, y, z;
public:
  cube(int i=0, int j=0, int k=0) {
    x=i;
    y=j;
    z=k;
  }

  int volume() {
    return x*y*z;
  }
};

int main()
{
  cube a(2,3,4), b;

  cout << a.volume() << endl;
  cout << b.volume();

  return 0;
}
```

There are two advantages to including default arguments, when appropriate, in a constructor. First, they prevent you from having to provide an overloaded constructor that takes no parameters. For example, if the parameters to **cube( )** were not given

defaults, the second constructor shown here would be needed to handle the declaration of **b** (which specified no arguments).

```
cube() {x=0; y=0; z=0}
```

Second, defaulting common initial values is more convenient than specifying them each time an object is declared.

Default Arguments vs. Overloading

In some situations, default arguments can be used as a shorthand form of function overloading. The **cube** class's constructor just shown is one example. Let's look at another. Imagine that you want to create two customized versions of the standard **strcat( )** function. The first version will operate like **strcat( )** and concatenate the entire contents of one string to the end of another. The second version takes a third argument that specifies the number of characters to concatenate. That is, the second version will only concatenate a specified number of characters from one string to the end of another. Thus, assuming that you call your customized functions **mystrcat( )**, they will have the following prototypes:

```
void mystrcat(char *s1, char *s2, int len);
void mystrcat(char *s1, char *s2);
```

The first version will copy **len** characters from **s2** to the end of **s1**. The second version will copy the entire string pointed to by **s2** onto the end of the string pointed to by **s1** and operates like **strcat( )**.

While it would not be wrong to implement two versions of **mystrcat( )** to create the two versions that you desire, there is an easier way. Using a default argument, you can create only one version of **mystrcat( )** that performs both functions. The following program demonstrates this.

```
// A customized version of strcat().
#include <iostream>
#include <cstring>
using namespace std;

void mystrcat(char *s1, char *s2, int len = -1);

int main()
{
  char str1[80] = "This is a test";
  char str2[80] = "0123456789";
```

```
  mystrcat(str1, str2, 5); // concatenate 5 chars
  cout << str1 << '\n';

  strcpy(str1, "This is a test"); // reset str1

  mystrcat(str1, str2); // concatenate entire string
  cout << str1 << '\n';

  return 0;
}

// A custom version of strcat().
void mystrcat(char *s1, char *s2, int len)
{
  // find end of s1
  while(*s1) s1++;

  if(len == -1) len = strlen(s2);

  while(*s2 && len) {
    *s1 = *s2; // copy chars
    s1++;
    s2++;
    len--;
  }

  *s1 = '\0'; // null terminate s1
}
```

Here, **mystrcat( )** concatenates up to **len** characters from the string pointed to by **s2** onto the end of the string pointed to by **s1**. However, if **len** is –1, as it will be when it is allowed to default, **mystrcat( )** concatenates the entire string pointed to by **s2** onto **s1**. (Thus, when **len** is –1, the function operates like the standard **strcat( )** function.) By using a default argument for **len**, it is possible to combine both operations into one function. In this way, default arguments sometimes provide an alternative to function overloading.

Using Default Arguments Correctly

Although default arguments can be a very powerful tool when used correctly, they can also be misused. The point of default arguments is to allow a function to perform its job in an efficient, easy-to-use manner while still allowing considerable flexibility. Toward

this end, all default arguments should reflect the way a function is generally used, or a reasonable alternate usage. When there is no single value that can be meaningfully associated with a parameter, there is no reason to declare a default argument. In fact, declaring default arguments when there is insufficient basis for doing so destructures your code, because they are liable to mislead and confuse anyone reading your program.

One other important guideline you should follow when using default arguments is this: No default argument should cause a harmful or destructive action. That is, the accidental use of a default argument should not cause a catastrophe.

Function Overloading and Ambiguity

You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions. When this happens, the situation is said to be *ambiguous*. Ambiguous statements are errors, and programs containing ambiguity will not compile.

By far the main cause of ambiguity involves C++'s automatic type conversions. As you know, C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function. For example, consider this fragment:

```
int myfunc(double d);
// ...
cout << myfunc('c'); // not an error, conversion applied
```

As the comment indicates, this is not an error because C++ automatically converts the character **c** into its **double** equivalent. In C++, very few type conversions of this sort are actually disallowed. Although automatic type conversions are convenient, they are also a prime cause of ambiguity. For example, consider the following program:

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

float myfunc(float i);
double myfunc(double i);

int main()
{
  cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
  cout << myfunc(10); // ambiguous

  return 0;
}
```

```
float myfunc(float i)
{
  return i;
}

double myfunc(double i)
{
  return -i;
}
```

Here, **myfunc( )** is overloaded so that it can take arguments of either type **float** or type **double**. In the unambiguous line, **myfunc(double)** is called because, unless explicitly specified as **float**, all floating-point constants in C++ are automatically of type **double**. Hence, that call is unambiguous. However, when **myfunc( )** is called by using the integer 10, ambiguity is introduced because the compiler has no way of knowing whether it should be converted to a **float** or to a **double**. This causes an error message to be displayed, and the program will not compile.

As the preceding example illustrates, it is not the overloading of **myfunc( )** relative to **double** and **float** that causes the ambiguity. Rather, it is the specific call to **myfunc( )** using an indeterminate type of argument that causes the confusion. Put differently, the error is not caused by the overloading of **myfunc( )**, but by the specific invocation.

Here is another example of ambiguity caused by C++'s automatic type conversions:

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

char myfunc(unsigned char ch);
char myfunc(char ch);

int main()
{
  cout << myfunc('c'); // this calls myfunc(char)
  cout << myfunc(88) << " "; // ambiguous

  return 0;
}

char myfunc(unsigned char ch)
{
  return ch-1;
```

```
}

char myfunc(char ch)
{
  return ch+1;
}
```

In C++, **unsigned char** and **char** are *not* inherently ambiguous. However, when **myfunc( )** is called by using the integer 88, the compiler does not know which function to call. That is, should 88 be converted into a **char** or an **unsigned char**?

Another way you can cause ambiguity is by using default arguments in overloaded functions. To see how, examine this program:

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

int myfunc(int i);
int myfunc(int i, int j=1);

int main()
{
  cout << myfunc(4, 5) << " "; // unambiguous
  cout << myfunc(10); // ambiguous

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

Here, in the first call to **myfunc( )**, two arguments are specified; therefore, no ambiguity is introduced and **myfunc(int i, int j)** is called. However, when the second call to **myfunc( )** is made, ambiguity occurs because the compiler does not know whether to call the version of **myfunc( )** that takes one argument or to apply the default to the version that takes two arguments.

Some types of overloaded functions are simply inherently ambiguous even if, at first, they may not seem so. For example, consider this program.

```
// This program contains an error.
#include <iostream>
using namespace std;

void f(int x);
void f(int &x); // error

int main()
{
  int a=10;

  f(a); // error, which f()?

  return 0;
}

void f(int x)
{
  cout << "In f(int)\n";
}

void f(int &x)
{
  cout << "In f(int &)\n";
}
```

As the comments in the program describe, two functions cannot be overloaded when the only difference is that one takes a reference parameter and the other takes a normal, call-by-value parameter. In this situation, the compiler has no way of knowing which version of the function is intended when it is called. Remember, there is no syntactical difference in the way that an argument is specified when it will be received by a reference parameter or by a value parameter.