Default Arguments and Function Overloading

All functions that receive arguments do not have to be sent values. C++ enables you to specify default argument lists. You can write functions that assume argument values even if you do not pass them any arguments.

C++ also enables you to write more than one function with the same function name. This is called *overloading functions.* As long as their argument lists differ, the functions are differentiated by C++.

This chapter introduces you to the following:

- Default argument lists
- Overloaded functions
- Name-mangling

Default argument lists and overloaded functions are not available in regular C. C++ extends the power of your programs by providing these time-saving procedures.

Default Argument Lists

Suppose you were writing a program that has to print a message on-screen for a short period of time. For instance, you pass a function an error message stored in a character array and the function prints the error message for a certain period of time.

The prototype for such a function can be this:

```
void pr_msg(char note[]);
```

Therefore, to request that `pr_msg()` print the line `"Turn printer on"`, you call it this way:

```
pr_msg("Turn printer on");    // Passes a message to be printed.
```

This command prints the message `"Turn printer on"` for a period of five seconds or so. To request that `pr_msg()` print the line `"Press any key to continue..."`, you call it this way:

```
pr_msg("Press a key to continue...");    // Passes a message.
```

As you write more of the program, you begin to realize that you are printing one message, for instance the `"Turn printer on"` message, more often than any other message. It seems as if the `pr_msg()` function is receiving that message much more often than any other. This might be the case if you were writing a program that printed many reports to the printer. You still will use `pr_msg()` for other delayed messages, but the `"Turn printer on"` message is most frequently used.

Instead of calling the function over and over, typing the same message each time, you can set up the prototype for `pr_msg()` so it defaults to the `"Turn printer on"` in this way:

```
void pr_msg(char note[]="Turn printer on");// Prototype
```

List default argument values in the prototype.

After prototyping `pr_msg()` with the default argument list, C++ assumes you want to pass `"Turn printer on"` to the function unless you override the default by passing something else to it. For instance, in `main()`, you call `pr_msg()` this way:

```
pr_msg();    // C++ assumes you mean "Turn printer on".
```

This makes your programming job easier. Because most of the time you want `pr_msg()` to print `"Turn printer on"` the default

argument list takes care of the message and you do not have to pass the message when you call the function. However, those few times when you want to pass something else, simply pass a different message. For example, to make `pr_msg()` print `"Incorrect value"` you type:

```
pr_msg("Incorrect value");    // Pass a new message.
```

TIP: Any time you call a function several times and find yourself passing that function the same parameters most of the time, consider using a default argument list.

Multiple Default Arguments

You can specify more than one default argument in the prototype list. Here is a prototype for a function with three default arguments:

```
float funct1(int i=10, float x=7.5, char c='A');
```

There are several ways you can call this function. Here are some samples:

```
funct1();
```

All default values are assumed.

```
funct1(25);
```

A `25` is sent to the integer argument, and the default values are assumed for the rest.

```
funct1(25, 31.25);
```

A `25` is sent to the integer argument, `31.25` to the floating-point argument, and the default value of `'A'` is assumed for the character argument.

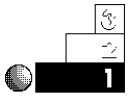
NOTE: If only some of a function's arguments are default arguments, those default arguments must appear on the far *left* of the argument list. No default arguments can appear to the left of those not specified as default. This is an *invalid* default argument prototype:

```
float func2(int i=10, float x, char c, long n=10.232);
```

This is invalid because a default argument appears on the left of a nondefault argument. To fix this, you have to move the two default arguments to the far left (the start) of the argument list. Therefore, by rearranging the prototype (and the resulting function calls) as follows, C++ enables you to accomplish the same objective as you attempted with the previous line:

```
float func2(float x, char c, int i=10, long n=10.232);
```

Examples



1. Here is a complete program that illustrates the message-printing function described earlier in this chapter. The `main()` function simply calls the delayed message-printing function three times, each time passing it a different set of argument lists.

```
// Filename: C20DEF1.CPP
// Illustrates default argument list.
#include <iostream.h>

void pr_msg(char note[]="Turn printer on"); // Prototype.

void main()
{
   pr_msg();                 // Prints default message.
   pr_msg("A new message");  // Prints another message.
   pr_msg();                 // Prints default message again.
   return;
}

void pr_msg(char note[])  // Only prototype contains defaults.
```

```
{
   long int delay;
   cout << note << "\n";
   for (delay=0; delay<500000; delay++)
      { ;  /* Do nothing while waiting */ }
   return;
}
```

The program produces the following output:

```
Turn printer on
A new message
Turn printer on
```

The delay loop causes each line to display for a couple of seconds or more, depending on the speed of your computer, until all three lines print.

2. The following program illustrates the use of defaulting several arguments. `main()` calls the function `de_fun()` five times, sending `de_fun()` five sets of arguments. The `de_fun()` function prints five different things depending on `main()`'s argument list.

```
// Filename: C20DEF2.CPP
// Demonstrates default argument list with several parameters.
#include <iostream.h>
#include <iomanip.h>

void de_fun(int i=5, long j=40034, float x=10.25,
            char ch='Z', double d=4.3234); // Prototype

void main()
{
   de_fun();          // All defaults used.
   de_fun(2);         // First default overridden.
   de_fun(2, 75037);  // First and second default overridden.
   de_fun(2, 75037, 35.88);      // First, second, and third
   de_fun(2, 75037, 35.88, 'G'); // First, second, third,
                                 // and fourth
   de_fun(2, 75037, 35.88, 'G', .0023); // No defaulting.
```

```
   return;
}

void de_fun(int i, long j, float x, char ch, double d)
{
   cout << setprecision(4) << "i: " << i << "    " << "j: " << j;
   cout << "    x: " << x << "    " << "ch: " << ch;
   cout << "    d: " << d << "\n";
   return;
}
```

Here is the output from this program:

```
i: 5    j: 40034    x: 10.25    ch: Z    d: 4.3234
i: 2    j: 40034    x: 10.25    ch: Z    d: 4.3234
i: 2    j: 75037    x: 10.25    ch: Z    d: 4.3234
i: 2    j: 75037    x: 35.88    ch: Z    d: 4.3234
i: 2    j: 75037    x: 35.88    ch: G    d: 4.3234
i: 2    j: 75037    x: 35.88    ch: G    d: 0.0023
```

Notice that each call to `de_fun()` produces a different output because `main()` sends a different set of parameters each time `main()` calls `de_fun()`.

Overloaded Functions

Unlike regular C, C++ enables you to have more than one function with the same name. In other words, you can have three functions called `abs()` in the same program. Functions with the same names are called overloaded functions. C++ requires that each overloaded function differ in its argument list. Overloaded functions enable you to have similar functions that work on different types of data.

For example, suppose you wrote a function that returned the absolute value of whatever number you passed to it. The absolute value of a number is its positive equivalent. For instance, the absolute value of 10.25 is 10.25 and the absolute value of –10.25 is 10.25.

Absolute values are used in distance, temperature, and weight calculations. The difference in the weights of two children is always

positive. If Joe weighs 65 pounds and Mary weighs 55 pounds, their difference is a positive 10 pounds. You can subtract the 65 from 55 (–10) or 55 from 65 (+10) and the weight difference is always the absolute value of the result.

Suppose you had to write an absolute-value function for integers, and an absolute-value function for floating-point numbers. Without function overloading, you need these two functions:

```
int iabs(int i)   // Returns absolute value of an integer.
{
   if (i < 0)
   {  return (i * -1); }  // Makes positive.
   else
   {  return (i);  }      // Already positive.
}

float fabs(float x)  // Returns absolute value of a float.
{
   if (x < 0.0)
   {  return (x * -1.0); }  // Makes positive.
   else
   {  return (x);  }        // Already positive.
}
```

Without overloading, if you had a floating-point variable for which you needed the absolute value, you pass it to the `fabs()` function as in:

```
ans = fabs(weight);
```

If you needed the absolute value of an integer variable, you pass it to the `iabs()` function as in:

```
ians = iabs(age);
```

Because the code for these two functions differ only in their parameter lists, they are perfect candidates for overloaded functions. Call both functions `abs()`, prototype both of them, and code each of them separately in your program. After overloading the two functions (each of which works on two different types of parameters with the same name), you pass your floating-point or integer value to `abs()`. The C++ compiler determines which function you wanted to call.

CAUTION: If two or more functions differ only in their return types, C++ cannot overload them. Two or more functions that differ only in their return types must have different names and cannot be overloaded.

This process simplifies your programming considerably. Instead of having to remember several different function names, you only have to remember one function name. C++ passes the arguments to the proper function.

NOTE: C++ uses *name-mangling* to accomplish overloaded functions. Understanding name-mangling helps you as you become an advanced C++ programmer.

When C++ realizes that you are overloading two or more functions with the same name, each function differing only in its parameter list, C++ changes the name of the function and adds letters to the end of the function name that match the parameters. Different C++ compilers do this differently.

To understand what the compiler does, take the absolute value function described earlier. C++ might change the integer absolute value function to `absi()` and the floating-point absolute value function to `absf()`. When you call the function with this function call:

```
ians = abs(age);
```

C++ determines that you want the `absi()` function called. As far as you know, C++ is not mangling the names; you never see the name differences in your program's source code. However, the compiler performs the name-mangling so it can keep track of different functions that have the same name.

Examples

1. Here is the complete absolute value program described in the previous text. Notice that both functions are prototyped. (The two prototypes signal C++ that it must perform name-mangling to determine the correct function names to call.)

```
// Filename: C20OVF1.CPP
// Overloads two absolute value functions.
#include <iostream.h>  // Prototype cout and cin.
#include <iomanip.h>   // Prototype setprecision(2).

int abs(int i);      // abs() is overloaded twice
float abs(float x);  // as shown by these prototypes.

void main()
{
   int ians;       // To hold return values.
   float fans;
   int i = -15;    // To pass to the two overloaded functions.
   float x = -64.53;

   ians = abs(i);    // C++ calls the integer abs().
   cout << "Integer absolute value of -15 is " << ians << "\n";

   fans = abs(x); // C++ calls the floating-point abs().
   cout << "Float absolute value of -64.53 is " <<
           setprecision(2) << fans << "\n";

   // Notice that you no longer have to keep track of two
   // different names. C++ calls the appropriate
   // function that matches the parameters.
   return;
}

int abs(int i)     // Integer absolute value function
{
   if (i < 0)
   {  return (i * -1); }  // Makes positive.
   else
   {  return (i);  }      // Already positive.
}
```

```
float abs(float x)  // Floating-point absolute value function
{
   if (x < 0.0)
   {  return (x * -1.0); }  // Makes positive.
   else
   {  return (x);  }        // Already positive.
}
```

The output from this program follows:

```
Integer absolute value of -15 is 15
Float absolute value of -64.53 is 64.53
```

2. As you write more and more C++ programs, you will see many uses for overloaded functions. The following program is a demonstration program showing how you can build your own output functions to suit your needs. `main()` calls three functions named `output()`. Each time it's called, `main()` passes a different value to the function.

   When `main()` passes `output()` a string, `output()` prints the string, formatted to a width (using the `setw()` manipulator described in Chapter 7, "Simple Input/Output") of 30 characters. When `main()` passes `output()` an integer, `output()` prints the integer with a width of five. When `main()` passes `output()` a floating-point value, `output()` prints the value to two decimal places and generalizes the output of different types of data. You do not have to format your own data. `output()` properly formats the data and you only have to remember one function name that outputs all three types of data.

```
// Filename: C20OVF2.CPP
// Outputs three different types of
// data with same function name.
#include <iostream.h>
#include <iomanip.h>
void output(char []);  // Prototypes for overloaded functions.
void output(int i);
void output(float x);
```

```
void main()
{
   char name[] = "C++ By Example makes C++ easy!";
   int ivalue = 2543;
   float fvalue = 39.4321;

   output(name);   // C++ chooses the appropriate function.
   output(ivalue);
   output(fvalue);

return;
}

void output(char name[])
{
   cout << setw(30) << name << "\n";
   // The width truncates string if it is longer than 30.
   return;
}

void output(int ivalue)
{
   cout << setw(5) << ivalue << "\n";
   // Just printed integer within a width of five spaces.
   return;
}

void output(float fvalue)
{
   cout << setprecision(2) << fvalue << "\n";
   // Limited the floating-point value to two decimal places.
   return;
}
```

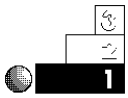
Here is the output from this program:

```
C++ By Example makes C++ easy!
2543
39.43
```

Each of the three lines, containing three different lines of information, was printed with the same function call.

Review Questions

The answers to the review questions are in Appendix B.



1. Where in the program do you specify the defaults for default argument lists?
2. What is the term for C++ functions that have the same name?

3. Does name-mangling help support default argument lists or overloaded functions?
4. True or false: You can specify only a single default argument.

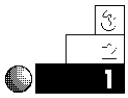
5. Fix the following prototype for a default argument list.

```
void my_fun(int i=7, float x, char ch='A');
```

6. True or false: The following prototypes specify overloaded functions:

```
int   sq_rt(int n);
float sq_rt(int n);
```

Review Exercises



1. Write a program that contains two functions. The first function returns the square of the integer passed to it, and the second function returns the square of the float passed to it.

2. Write a program that computes net pay based on the values the user types. Ask the user for the hours worked, the rate per hour, and the tax rate. Because the majority of employees work 40 hours per week and earn $5.00 per hour, use these values as default values in the function that computes the net pay. If the user presses Enter in response to your questions, use the default values.

Summary

Default argument lists and overloaded functions speed up your programming time. You no longer have to specify values for common arguments. You do not have to remember several different names for those functions that perform similar routines and differ only in their data types.

The remainder of this book elaborates on earlier concepts so you can take advantage of separate, modular functions and local data. You are ready to learn more about how C++ performs input and output. Chapter 21, "Device and Character Input/Output," teaches you the theory behind I/O in C++, and introduces more built-in functions.