

# Module6

## A Closer Look at Functions

### **Table of Contents**

CRITICAL SKILL 6.1: Know the two approaches to argument passing.....	2
CRITICAL SKILL 6.2: How C++ Passes Arguments .....	2
CRITICAL SKILL 6.3: Using a Pointer to Create a Call-by-Reference .....	3
CRITICAL SKILL 6.4: Reference Parameters .....	4
CRITICAL SKILL 6.5: Returning References .....	9
CRITICAL SKILL 6.6: Independent References .....	12
CRITICAL SKILL 6.7: Function Overloading .....	13
CRITICAL SKILL 6.8: Default Function Arguments .....	26
CRITICAL SKILL 6.9: Function Overloading and Ambiguity .....	29

This module continues our examination of the function. It discusses three of C++'s most important function-related topics: references, function overloading, and default arguments. These features vastly expand the capabilities of a function. A reference is an implicit pointer. Function overloading is the quality that allows one function to be implemented two or more different ways, each performing a separate task. Function overloading is one way that C++ supports polymorphism. Using a default argument, it is possible to specify a value for a parameter that will be automatically used when no corresponding argument is specified. We will begin with an explanation of the two ways that arguments can be passed to functions, and the implications of both methods. An understanding of argument passing is needed in order to understand the reference.

## CRITICAL SKILL 6.1: Know the two approaches to argument passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first is call-by-value. This method copies the value of an argument into the parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument used to call it.

Call-by-reference is the second way a subroutine can be passed arguments. In this method, the address of an argument (not its value) is copied into the parameter. Inside the subroutine, this address is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

## CRITICAL SKILL 6.2: How C++ Passes Arguments

By default, C++ uses call-by-value for passing arguments. This means that the code inside a function cannot alter the arguments used to call the function. In this book, all of the programs up to this point have used the call-by-value method. For example, consider the `reciprocal()` function in this program:

```
// Changing a call-by-value parameter does not affect the
// argument.

#include <iostream>
using namespace std;

double reciprocal(double x);

int main()
{
    double t = 10.0;

    cout << "Reciprocal of 10.0 is " << reciprocal(t) << "\n";

    cout << "Value of t is still: " << t << "\n";

    return 0;
}

// Return the reciprocal of a value.
double reciprocal(double x)
{
    x = 1 / x; // create reciprocal ← This does not change the
                                     value of t inside main().

    return x;
}
```

takes place inside `reciprocal()`, the only thing modified is the local variable `x`. The variable `t` used as an argument will still have the value 10 and is unaffected by the operations inside the function.

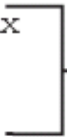
## CRITICAL SKILL 6.3: Using a Pointer to Create a Call-by-Reference

Even though C++'s default parameter-passing convention is call-by-value, it is possible to manually create a call-by-reference by passing the address of an argument (that is, a pointer) to a function. It is then possible to change the value of the argument outside of the function. You saw an example of this in the preceding module when the passing of pointers was discussed. As you know, pointers are passed to functions just like any other values. Of course, it is necessary to declare the parameters as pointer types.

To see how passing a pointer allows you to manually create a call-by-reference, consider a function called `swap()` that exchanges the values of the two variables pointed to by its arguments. Here is one way to implement it:

```
// Exchange the values of the variables pointed to by x and y.
void swap(int *x, int *y)
{
    int temp;

    temp = *x; // save the value at address x
    *x = *y;    // put y into x
    *y = temp; // put x into y
}
```



Exchange the values of the variables pointed to by x and y.

The `swap()` function declares two pointer parameters, `x` and `y`. It uses these parameters to exchange the values of the variables pointed to by the arguments passed to the function. Remember, `*x` and `*y` refer to the variables pointed to by `x` and `y`. Thus, the statement

```
*x = *y;
```

puts the value of the object pointed to by `y` into the object pointed to by `x`. Consequently, when the function terminates, the contents of the variables used to call the function will be swapped.

Since `swap()` expects to receive two pointers, you must remember to call `swap()` with the addresses of the variables you want to exchange. The correct method is shown in this program:

```
// Demonstrate the pointer version of swap().

#include <iostream>
using namespace std;

// Declare swap() using pointers.
void swap(int *x, int *y);
```

```

int main()
{
    int i, j;

    i = 10;
    j = 20;

    cout << "Initial values of i and j: ";
    cout << i << ' ' << j << '\n';

    swap(&j, &i); // call swap() with addresses of i and j

    cout << "Swapped values of i and j: ";
    cout << i << ' ' << j << '\n';

    return 0;
}

```

Call **swap()** with the addresses of the variables you want to exchange.

In `main()`, the variable `i` is assigned the value 10, and `j`, the value 20. Then `swap()` is called with the addresses of `i` and `j`. The unary operator `&` is used to produce the addresses of the variables. Therefore, the addresses of `i` and `j`, not their values, are passed into `swap()`. When `swap()` returns, `i` and `j` will have their values exchanged, as the following output shows:

Initial values of i and j: 10 20 Swapped values of i and j: 20 10



1. Explain call-by-value.
2. Explain call-by-reference.
3. What parameter-passing mechanism does C++ use by default?

## CRITICAL SKILL 6.4: Reference Parameters

While it is possible to achieve a call-by-reference manually by using the pointer operators, this approach is rather clumsy. First, it compels you to perform all operations through pointers. Second, it requires that you remember to pass the addresses (rather than the values) of the arguments when calling the function. Fortunately, in C++, it is possible to tell the compiler to automatically use call-by-reference rather than call-by-value for one or more parameters of a particular function. You can accomplish this with a reference parameter. When you use a reference parameter, the address (not the value) of an argument is automatically passed to the function. Within the function, operations on the reference parameter are automatically dereferenced, so there is no need to use the pointer operators.

A reference parameter is declared by preceding the parameter name in the function's declaration with an &. Operations performed on a reference parameter affect the argument used to call the function, not the reference parameter itself.

To understand reference parameters, let's begin with a simple example. In the following, the function `f()` takes one reference parameter of type `int`:

```
// Using a reference parameter.

#include <iostream>
using namespace std;

void f(int &i); // here, i is a reference parameter ←
               |
               | Declare a reference
               | parameter.

int main()
{
    int val = 1;

    cout << "Old value for val: " << val << '\n';

    f(val); // pass address of val to f()

    cout << "New value for val: " << val << '\n';

    return 0;
}

void f(int &i)
{
    i = 10; // this modifies calling argument ←
           |
           | Assign a value to the
           | variable referred to by i.
}
```

This program displays the following output:

Old value for val: 1

New value for val: 10

Pay special attention to the definition of `f()`, shown here:

```
void f(int &i) {
    i = 10; // this modifies calling argument }
```

Notice the declaration of `i`. It is preceded by an `&`, which causes it to become a reference parameter. (This declaration is also used in the function's prototype.) Inside the function, the following statement

```
i = 10;
```

does not cause `i` to be given the value 10. Instead, it causes the variable referenced by `i` (in this case, `val`) to be assigned the value 10. Notice that this statement does not use the `*` pointer operator. When you

use a reference parameter, the C++ compiler automatically knows that it is an address and dereferences it for you. In fact, using the `*` would be an error.

Since `i` has been declared as a reference parameter, the compiler will automatically pass `f()` the address of any argument it is called with. Thus, in `main()`, the statement

```
// Use reference parameters to create the swap() function.

#include <iostream>
using namespace std;

// Declare swap() using reference parameters.
void swap(int &x, int &y);

int main()
{
    int i, j;

    i = 10;
    j = 20;

    cout << "Initial values of i and j: ";

    cout << i << ' ' << j << '\n';

    swap(j, i); ← Here, the addresses of i and j are automatically passed to swap().

    cout << "Swapped values of i and j: ";
    cout << i << ' ' << j << '\n';

    return 0;
}

/* Here, swap() is defined as using call-by-reference,
   not call-by-value. Thus, it can exchange the two
   arguments it is called with.
*/
void swap(int &x, int &y)
{
    int temp;

    // use references to exchange the values of the arguments
    temp = x;
    x = y; ← Now, the exchange takes place automatically through the references.
    y = temp;
}
```

passes the address of `val` (not its value) to `f( )`. There is no need to precede `val` with the `&` operator. (Doing so would be an error.) Since `f( )` receives the address of `val` in the form of a reference, it can modify the value of `val`.

To illustrate reference parameters in actual use—and to fully demonstrate their benefits—the `swap( )` function is rewritten using references in the following program. Look carefully at how `swap( )` is declared and called.

```
// Use reference parameters to create the swap() function.

#include <iostream> using namespace std;

// Declare swap() using reference parameters. void swap(int &x, int &y);

int main() {

    int i, j;

    i = 10;

    j = 20;

    cout << "Initial values of i and j: ";

    /* Here, swap() is defined as using call-by-reference,
    not call-by-value. Thus, it can exchange the two
    arguments it is called with. */ void swap(int &x, int &y) { int temp;

    // use references to exchange the values of the arguments temp = x; x = y;

    Now, the exchange takes place y = temp; automatically through the references. }
```

The output is the same as the previous version. Again, notice that by making `x` and `y` reference parameters, there is no need to use the `*` operator when exchanging values. Remember, the compiler automatically generates the addresses of the arguments used to call `swap( )` and automatically dereferences `x` and `y`.

Let's review. When you create a reference parameter, that parameter automatically refers to (that is, implicitly points to) the argument used to call the function. Further, there is no need to apply the `&` operator to an argument. Also, inside the function, the reference parameter is used directly; the `*` operator is not used. All operations involving the reference parameter automatically refer to the argument used in the call to the function. Finally, when you assign a value to a reference parameter, you are actually assigning that value to the variable to which the reference is pointing. In the case of a function parameter, this will be the variable used in the call to the function.

One last point: The C language does not support references. Thus, the only way to create a call-by-reference in C is to use pointers, as shown earlier in the first version of `swap( )`. When converting C code to C++, you will want to convert these types of parameters to references, where feasible.

# Ask the Expert

**Q:** In some C++ code, I have seen a declaration style in which the & is associated with the type name as shown here:

```
int& i;
```

rather than the variable name, like this:

```
int &i;
```

Is there a difference?

**A:** The short answer is no, there is no difference between the two declarations. For example, here is another way to write the prototype to swap( ):

```
void swap(int& x, int& y);
```

As you can see, the & is immediately adjacent to int and not to x. Furthermore, some programmers also specify pointers by associating the \* with the type rather than the variable, as shown here:

```
float* p;
```

These types of declarations reflect the desire by some programmers for C++ to contain a separate reference or pointer type. However, the trouble with associating the & or \* with the type rather than the variable is that, according to the formal C++ syntax, neither the & nor the \* is distributive over a list of variables, and this can lead to confusing declarations. For example, the following declaration creates one, not two, int pointers:

```
int* a, b;
```

Here, b is declared as an integer (not an integer pointer) because, as specified by the C++ syntax, when used in a declaration, an \* or an & is linked to the individual variable that it precedes, not to the type that it follows.

It is important to understand that as far as the C++ compiler is concerned, it doesn't matter whether you write `int *p` or `int* p`. Thus, if you prefer to associate the \* or & with the type rather than the variable, feel free to do so. However, to avoid confusion, this book will continue to associate the \* and the & with the variable name that each modifies, rather than with the type name.



1. How is a reference parameter declared?



2. When calling a function that uses a reference parameter, must you precede the argument with an `&`?
3. Inside a function that receives a reference parameter, do operations on that parameter need to be preceded with an `*` or `&`?

## CRITICAL SKILL 6.5: Returning References

A function can return a reference. In C++ programming, there are several uses for reference return values. Some of these uses must wait until later in this book. However, there are some that you can use now.

When a function returns a reference, it returns an implicit pointer to its return value. This gives rise to a rather startling possibility: the function can be used on the left side of an assignment statement! For example, consider this simple program:

```
// Returning a reference.

#include <iostream>
using namespace std;

double &f(); // return a reference. ← Here, f() returns a
                                     reference to a double.

double val = 100.0;

int main()
{
    double x;

    cout << f() << '\n'; // display val's value

    x = f(); // assign value of val to x
    cout << x << '\n'; // display x's value

    f() = 99.1; // change val's value
    cout << f() << '\n'; // display val's new value

    return 0;
}

// This function returns a reference to a double.
double &f()
{
    return val; // return reference to val ← This returns a reference to
                                                the global variable val.
}
```

The output of this program is shown here:

```
100
100
99.1
```

Let's examine this program closely. At the beginning, `f()` is declared as returning a reference to a double, and the global variable `val` is initialized to 100. In `main()`, the following statement displays the original value of `val`:

```
cout << f() << '\n'; // display val's value
```

When `f()` is called, it returns a reference to `val` using this return statement:

```
return val; // return reference to val
```

This statement automatically returns a reference to `val` rather than `val`'s value. This reference is then used by the `cout` statement to display `val`'s value.

In the line

```
x = f(); // assign value of val to x
```

the reference to `val` returned by `f()` assigns the value of `val` to `x`. The most interesting line in the program is shown here:

```
f() = 99.1; // change val's value
```

This statement causes the value of `val` to be changed to 99.1. Here is why: since `f()` returns a reference to `val`, this reference becomes the target of the assignment statement. Thus, the value of 99.1 is assigned to `val` indirectly, through the reference to it returned by `f()`.

Here is another sample program that uses a reference return type:

```

// Return a reference to an array element.

#include <iostream>
using namespace std;

double &change_it(int i); // return a reference

double vals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

int main()
{
    int i;

    cout << "Here are the original values: ";
    for(i=0; i < 5; i++)
        cout << vals[i] << ' ';
    cout << '\n';

    change_it(1) = 5298.23; // change 2nd element
    change_it(3) = -98.8; // change 4th element

    cout << "Here are the changed values: ";
    for(i=0; i < 5; i++)
        cout << vals[i] << ' ';
    cout << '\n';

    return 0;
}

double &change_it(int i)
{
    return vals[i]; // return a reference to the ith element
}

```

This program changes the values of the second and fourth elements in the vals array. The program displays the following output:

```

Here are the original values: 1.1 2.2 3.3 4.4 5.5
Here are the changed values: 1.1 5298.23 3.3 -98.8 5.5

```

Let's see how this is accomplished.

The `change_it()` function is declared as returning a reference to a double. Specifically, it returns a reference to the element of `vals` that is specified by its parameter `i`. The reference returned by `change_it()` is then used in `main()` to assign a value to that element.

When returning a reference, be careful that the object being referred to does not go out of scope. For example, consider this function:

```
// Error, cannot return reference to local var.
int &f()
{
    int i = 10;

    return i; ← Error! i will go out-of-scope
               when f() returns.
}
```

In `f()`, the local variable `i` will go out of scope when the function returns. Therefore, the reference to `i` returned by `f()` will be undefined. Actually, some compilers will not compile `f()` as written for precisely this reason. However, this type of problem can be created indirectly, so be careful which object you return a reference to.

## CRITICAL SKILL 6.6: Independent References

Even though the reference is included in C++ primarily for supporting call-by-reference parameter passing and for use as a function return type, it is possible to declare a stand-alone reference variable. This is called an independent reference. It must be stated at the outset, however, that non-parameter reference variables are seldom used, because they tend to confuse and destructure your program. With these reservations in mind, we will take a short look at them here.

An independent reference must point to some object. Thus, an independent reference must be initialized when it is declared. Generally, this means that it will be assigned the address of a previously declared variable. Once this is done, the name of the reference variable can be used anywhere that the variable it refers to can be used. In fact, there is virtually no distinction between the two. For example, consider the program shown here:

```
// Use an independent reference.

#include <iostream>

using namespace std;

int main()
{
    int j, k;
    int &i = j; // independent reference ← An independent reference

    j = 10;

    cout << j << " " << i; // outputs 10 10

    k = 121;
    i = k; // copies k's value into j, not k's address

    cout << "\n" << j; // outputs 121

    return 0;
}
```

This program displays the following output:

```
10 10
```

```
121
```

The address pointed to by a reference variable is fixed; it cannot be changed. Thus, when the statement

```
i = k;
```

is evaluated, it is k's value that is copied into j (referred to by i), not its address.

As stated earlier, it is generally not a good idea to use independent references, because they are not necessary and they tend to garble your code. Having two names for the same variable is an inherently confusing situation.

## A Few Restrictions When Using References

- There are some restrictions that apply to reference variables:
- You cannot reference a reference variable.
- You cannot create arrays of references.
- You cannot create a pointer to a reference. That is, you cannot apply the & operator to a reference.



1. Can a function return a reference?
2. What is an independent reference?
3. Can you create a reference to a reference?

## CRITICAL SKILL 6.7: Function Overloading

In this section, you will learn about one of C++'s most exciting features: function overloading. In C++, two or more functions can share the same name as long as their parameter declarations are different. In this situation, the functions that share the same name are said to be overloaded, and the process is referred to as function overloading. Function overloading is one way that C++ achieves polymorphism.

In general, to overload a function, simply declare different versions of it. The compiler takes care of the rest. You must observe one important restriction: the type and/or number of the parameters of each overloaded function must differ. It is not sufficient for two functions to differ only in their return types. They must differ in the types or number of their parameters. (Return types do not provide sufficient information in all cases for C++ to decide which function to use.) Of course, overloaded functions may differ in their return types, too. When an overloaded function is called, the version of the function whose parameters match the arguments is executed.

Let's begin with a short sample program:

```
// Overload a function three times.

#include <iostream>
using namespace std;

void f(int i);           // integer parameter
void f(int i, int j);    // two integer parameters
void f(double k);        // one double parameter

int main()
{
    f(10);               // call f(int)

    f(10, 20);           // call f(int, int)

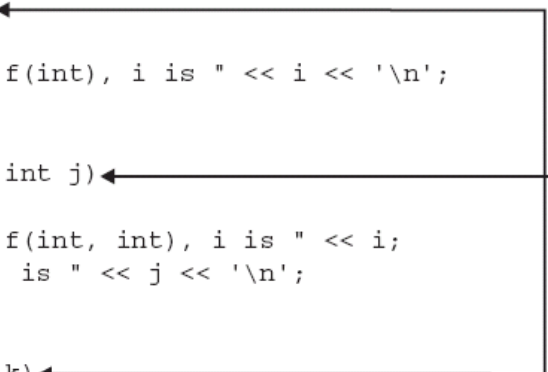
    f(12.23);            // call f(double)

    return 0;
}

void f(int i)
{
    cout << "In f(int), i is " << i << '\n';
}

void f(int i, int j)
{
    cout << "In f(int, int), i is " << i;
    cout << ", j is " << j << '\n';
}

void f(double k)
{
    cout << "In f(double), k is " << k << '\n';
}
```



A separate prototype is needed for each version of an overloaded function.

Here are the three different implementations of f().

This program produces the following output:

```
In f(int), i is 10
In f(int, int), i is 10, j is 20
In f(double), k is 12.23
```

As you can see, `f()` is overloaded three times. The first version takes one integer parameter, the second version requires two integer parameters, and the third version has one double parameter. Because the parameter list for each version is different, the compiler is able to call the correct version of each function based on the type of the arguments specified at the time of the call. To understand the value of function overloading, consider a function called `neg()` that returns the negation of its arguments. For example, when called with the value `-10`, `neg()` returns `10`. When called with `9`, it returns `-9`. Without

function overloading, if you wanted to create negation functions for data of type int, double, and long, you would need three different functions, each with a different name, such as `ineg()`, `lneg()`, and `fneg()`. However, through the use of function overloading, you can use one name, such as `neg()`, to refer to all functions that return the negation of their argument. Thus, overloading supports the polymorphic concept of “one interface, multiple methods.” The following program demonstrates this:

```
// Create various versions of the neg() function.

#include <iostream>
using namespace std;

int neg(int n);          // neg() for int.
double neg(double n);    // neg() for double.
long neg(long n);        // neg() for long.

int main()
{
    cout << "neg(-10): " << neg(-10) << "\n";
    cout << "neg(9L): " << neg(9L) << "\n";
    cout << "neg(11.23): " << neg(11.23) << "\n";

    return 0;
}

// neg() for int.
int neg(int n)
{
    return -n;
}

// neg() for double.
double neg(double n)
{
    return -n;
}

// neg() for long.
long neg(long n)
{
    return -n;
}
```

The output is shown here:

```
neg(-10): 10
neg(9L): -9
```

neg(11.23): -11.23

This program creates three similar but different functions called `neg`, each of which returns the absolute value of its argument. The compiler knows which function to use in each given situation because of the type of the argument.

The value of overloading is that it allows related sets of functions to be accessed using a common name. Thus, the name `neg` represents the general action that is being performed. It is left to the compiler to choose the right specific version for a particular circumstance. You, the programmer, need only remember the general action being performed. Therefore, through the application of polymorphism, three things to remember have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.

Another advantage to function overloading is that it is possible to define slightly different versions of the same function that are specialized for the type of data upon which they operate. For example, consider a function called `min()` that determines the minimum of two values. It is possible to create versions of `min()` that behave differently for different data types. When comparing two integers, `min()` returns the smallest integer. When two characters are compared, `min()` could return the letter that is first in alphabetical order, ignoring case differences. In the ASCII sequence, uppercase characters are represented by values that are 32 less than the lowercase letters. Thus, ignoring case would be useful when alphabetizing. When comparing two pointers, it is possible to have `min()` compare the values pointed to by the pointers and return the pointer to the smallest value. Here is a program that implements these versions of `min()`:

```
// Create various versions of min().

#include <iostream>
using namespace std;

int min(int a, int b);      // min() for ints
char min(char a, char b);  // min() for chars
int * min(int *a, int *b); // min() for int pointers

int main()
{
    int i=10, j=22;

    cout << "min('X', 'a'): " << min('X', 'a') << "\n";
    cout << "min(9, 3): " << min(9, 3) << "\n";
    cout << "*min(&i, &j): " << *min(&i, &i) << "\n";

    return 0;
}

// min() for ints. Return the smallest value.
int min(int a, int b) ← Each version of min() can be
{                       specialized. This version
    if(a < b) return a;  returns the smallest int value.
    else return b;
}
```



```

}

// min() for chars -- ignore case.
char min(char a, char b) ← This version of min()
                           ignores case.
{
    if(tolower(a) < tolower(b)) return a;
    else return b;
}

/*
   min() for int pointers.
   Compare values and return pointer to smallest value.
*/
int * min(int *a, int *b) ← This version of min()
                           returns a pointer to
                           the smallest value.
{
    if(*a < *b) return a;
    else return b;
}

```

Here is the output produced by the program:

```

min('X', 'a'): a
min(9, 3): 3
*min(&i, &j): 10

```

When you overload a function, each version of that function can perform any activity you desire. That is, there is no rule stating that overloaded functions must relate to one another. However, from a stylistic point of view, function overloading implies a relationship. Thus, while you can use the same name to overload unrelated functions, you should not. For example, you could use the name `sqr( )` to create functions that return the square of an `int` and the square root of a `double`. These two operations are fundamentally different, however, and applying function overloading in this manner defeats its original purpose. (In fact, programming in this manner is considered to be extremely bad style!) In practice, you should overload only closely related operations.

## Automatic Type Conversions and Overloading

As you will recall from Module 2, C++ provides certain automatic type conversions. These conversions also apply to parameters of overloaded functions. For example, consider the following:

```

/*
   Automatic type conversions can affect
   overloaded function resolution.

```

```

*/

#include <iostream>
using namespace std;

void f(int x);

void f(double x);

int main() {
    int i = 10;
    double d = 10.1;
    short s = 99;
    float r = 11.5F;

    f(i); // calls f(int)
    f(d); // calls f(double)

    f(s); // calls f(int) -- type conversion
    f(r); // calls f(double) -- type conversion
}

return 0;
}

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

void f(double x) {
    cout << "Inside f(double): " << x << "\n";
}

```

Automatic type conversions occur here.

The output from the program is shown here:

```

Inside f(int): 10
Inside f(double): 10.1
Inside f(int): 99
Inside f(double): 11.5

```

In this example, only two versions of `f()` are defined: one that has an `int` parameter and one that has a `double` parameter. However, it is possible to pass `f()` a `short` or `float` value. In the case of `short`, C++ automatically converts it to `int`. Thus, `f(int)` is invoked. In the case of `float`, the value is converted to `double` and `f(double)` is called.

It is important to understand, however, that the automatic conversions apply only if there is no direct match between a parameter and an argument. For example, here is the preceding program with the addition of a version of `f()` that specifies a `short` parameter:

```

// Now, add f(short).

#include <iostream>
using namespace std;

void f(int x);
void f(short x); ← f(short) has
void f(double x);    been added.

int main() {
    int i = 10;
    double d = 10.1;
    short s = 99;
    float r = 11.5F;

    f(i); // calls f(int)
    f(d); // calls f(double)

    f(s); // now calls f(short) ← Now, no type conversion occurs
                                         because f(short) is available.

    f(r); // calls f(double) -- type conversion

    return 0;
}

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

void f(short x) {
    cout << "Inside f(short): " << x << "\n";
}

void f(double x) {
    cout << "Inside f(double): " << x << "\n";
}

```

Now when the program is run, the following output is produced:

```

Inside f(int): 10
Inside f(double): 10.1
Inside f(short): 99
Inside f(double): 11.5

```

In this version, since there is a version of `f()` that takes a short argument, when `f()` is called with a short value, `f(short)` is invoked and the automatic conversion to `int` does not occur.

## Progress Check

1. When a function is overloaded, what condition must be met?
2. Why should overloaded functions perform related actions?
3. Does the return type of a function participate in overload resolution?

### Project 6-1 Create Overloaded Output Functions

In this project, you will create a collection of overloaded functions that output various data types to the screen. Although using `cout` statements is quite convenient, such a collection of output functions offers an alternative that might appeal to some programmers. In fact, both Java and C# use output functions rather than output operators. By creating overloaded output functions, you can use either method and have the best of both worlds. Furthermore, you can tailor your output functions to meet your specific needs. For example, you can make the Boolean values display “true” or “false” rather than 1 and 0.

You will be creating two sets of functions called `println( )` and `print( )`. The `println( )` function displays its argument followed by a newline. The `print( )` function will display its argument, but does not append a newline. For example,

```
print(1);
println('X');
print("Function overloading is powerful. ");
print(18.22);
```

displays

1X

Function overloading is powerful. 18.22

In this project, `print( )` and `println( )` will be overloaded for data of type `bool`, `char`, `int`, `long`, `char *`, and `double`, but you can add other types on your own.

## Step by Step

1. Create a file called `Print.cpp`.

**2. Begin the project with these lines:**

```
/*
    Project 6-1

    Create overloaded print() and println() functions
    that display various types of data.
*/
include <iostream>
using namespace std;
```

**3. Add the prototypes for the print( ) and println( ) functions, as shown here:**

```
// These output a newline.
void println(bool b);
void println(int i);
void println(long i);
void println(char ch);
void println(char *str);
void println(double d);

// These functions do not output a newline.
void print(bool b);
void print(int i);
void print(long i);
void print(char ch);
void print(char *str);
void print(double d);
```

**4. Implement the println( ) functions, as shown here:**

```
// Here are the println() functions.
void println(bool b)
{
    if(b) cout << "true\n";
    else cout << "false\n";
}

void println(int i)
{
    cout << i << "\n";
}
```

```

}

void println(long i)
{
    cout << i << "\n";
}

void println(char ch)
{
    cout << ch << "\n";
}

void println(char *str)
{
    cout << str << "\n";
}

void println(double d)
{
    cout << d << "\n";
}

```

Notice that each function appends a newline character to the output. Also notice that `println(bool)` displays either “true” or “false” when a Boolean value is output. This illustrates how you can easily customize output to meet your own needs and tastes.

5. Implement the `print()` functions, as shown next:

```

// Here are the print() functions
void print(bool b)
{
    if(b) cout << "true";
    else cout << "false";
}

void print(int i)
{
    cout << i;
}

void print(long i)
{
    cout << i;
}

void print(char ch)
{

```

```

    cout << ch;
}

void print(char *str)
{
    cout << str;
}

void print(double d)
{
    cout << d;
}

```

These functions are the same as their `println()` counterparts except that they do not output a newline. Thus, subsequent output appears on the same line.

6. Here is the complete `Print.cpp` program:

```

/*
    Project 6-1

    Create overloaded print() and println() functions
    that display various types of data.
*/

#include <iostream>
using namespace std;

// These output a newline.
void println(bool b);
void println(int i);
void println(long i);
void println(char ch);
void println(char *str);
void println(double d);

// These functions do not output a newline.
void print(bool b);
void print(int i);
void print(long i);
void print(char ch);
void print(char *str);
void print(double d);

int main()
{
    println(true);
}

```

```

    println(10);
    println("This is a test");
    println('x');
    println(99L);
    println(123.23);

    print("Here are some values: ");
    print(false);
    print(' ');
    print(88);
    print(' ');
    print(100000L);
    print(' ');
    print(100.01);

    println(" Done!");

    return 0;
}

// Here are the println() functions.
void println(bool b)
{
    if(b) cout << "true\n";
    else cout << "false\n";
}

void println(int i)
{
    cout << i << "\n";
}

void println(long i)
{
    cout << i << "\n";
}

void println(char ch)
{
    cout << ch << "\n";
}

void println(char *str)
{
    cout << str << "\n";
}
- }
```



```

void println(double d)
{
    cout << d << "\n";
}

// Here are the print() functions.
void print(bool b)
{
    if(b) cout << "true";
    else cout << "false";
}

void print(int i)
{
    cout << i;
}

void print(long i)
{
    cout << i;
}

void print(char ch)
{
    cout << ch;
}

void print(char *str)
{
    cout << str;
}

void print(double d)
{
    cout << d;
}

```

The output from the program is shown here:

```

true
10
This is a test
x
99
123.23
Here are some values: false 88 100000 100.01 Done!

```

## CRITICAL SKILL 6.8: Default Function Arguments

The next function-related feature to be discussed is the default argument. In C++, you can give a parameter a default value that is automatically used when no argument corresponding to that parameter is specified in a call to a function. Default arguments can be used to simplify calls to complex functions. Also, they can sometimes be used as a “shorthand” form of function overloading.

A default argument is specified in a manner syntactically similar to a variable initialization. Consider the following example, which declares `myfunc( )` as taking two `int` arguments. The first defaults to 0. The second defaults to 100.

```
void myfunc(int x = 0, int y = 100);
```

Now `myfunc( )` can be called by one of the three methods shown here:

```
myfunc(1, 2); // pass explicit values
```

```
myfunc(10); // pass x a value, let y default
```

```
myfunc(); // let both x and y default
```

The first call passes the value 1 to `x` and 2 to `y`. The second gives `x` the value 10 and allows `y` to default to 100. Finally, the third call causes both `x` and `y` to default. The following program

```
// Demonstrate default arguments.

#include <iostream>
using namespace std;

void myfunc(int x = 0, int y = 100); ← myfunc( ) specifies
                                     default arguments for
                                     both parameters.

int main()
{
    myfunc(1, 2);

    myfunc(10);

    myfunc();

    return 0;
}

void myfunc(int x, int y)

{
    cout << "x: " << x << ", y: " << y << "\n";
}
```

The output shown here confirms the use of the default arguments:

```
1, y: 2
```

```
10, y: 100
```

```
0, y: 100
```

When creating a function that has default argument values, the default values must be specified only once, and this must happen the first time the function is declared within the file. In the preceding example, the default argument was specified in `myfunc( )`'s prototype. If you try to specify new (or even the same) default values in `myfunc( )`'s definition, the compiler will display an error message and will not compile your program.

Even though default arguments cannot be redefined within a program, you can specify different default arguments for each version of an overloaded function; that is, different versions of the overloaded function can have different default arguments.

It is important to understand that all parameters that take default values must appear to the right of those that do not. For example, the following prototype is invalid:

```
// Wrong! void f(int a = 1, int b);
```

Once you've begun defining parameters that take default values, you cannot specify a nondefaulting parameter. That is, a declaration like the following is also wrong and will not compile:

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

Since `i` has been given a default value, `j` must be given one, too.

One reason that default arguments are included in C++ is that they enable the programmer to manage greater complexity. To handle the widest variety of situations, quite frequently a function will contain more parameters than are required for its most common usage. Thus, when the default arguments apply, you need to remember and specify only the arguments that are meaningful to the exact situation, not all those needed for the most general case.

## Default Arguments Versus Overloading

One application of default arguments is as a shorthand form of function overloading. To see why this is the case, imagine that you want to create two customized versions of the standard `strcat( )` function. One version will operate like `strcat( )` and concatenate the entire contents of one string to the end of another. The other version will take a third argument that specifies the number of characters to concatenate. That is, this second version will concatenate only 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 will operate 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 operations. 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 = 0); ← Here, len defaults to zero.

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

    mystrcat(str1, str2, 5); // concatenate 5 chars ← Here, len is specified,
    cout << str1 << '\n';                                     and only 5 characters
                                                                are copied.

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

    mystrcat(str1, str2); // concatenate entire string ← Here, len defaults to
    cout << str1 << '\n';                                     zero, and the entire
                                                                string is concatenated.

    return 0;
}

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

    if(len==0) len = strlen(s2);

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

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

The output from the program is shown here:

This is a test01234 this is a test0123456789

As the program illustrates, `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 zero, 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 zero, 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. As this example illustrates, default arguments sometimes provide a shorthand form of function overloading.

## Using Default Arguments Correctly

Although default arguments are a 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 is normally associated with a parameter, then 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. Finally, a default argument should cause no harm. That is, the accidental use of a default argument should not have irreversible, negative consequences. For example, forgetting to specify an argument should not cause an important data file to be erased!



1. Show how to declare a void function called `count( )` that takes two `int` parameters called `a` and `b`, and give each a default value of 0.
2. Can default arguments be declared in both a function's prototype and its definition?
3. Is this declaration correct? If not, why not?  
`int f(int x=10, double b);`

## CRITICAL SKILL 6.9: Function Overloading and Ambiguity

Before concluding this module, it is necessary to discuss a type of error unique to C++: ambiguity. It is possible to create a situation in which the compiler is unable to choose between two (or more) correctly 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. C++ automatically attempts to convert the type of the arguments used to call a function into the type of the parameters defined by the function. Here is an example:

```
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. Actually, in C++, very few type conversions of this sort are disallowed. While automatic type conversions are convenient, they are also a prime cause of ambiguity. Consider the following program:

```
// Overloading ambiguity.

#include <iostream>
using namespace std;

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

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

    // ambiguous
    cout << myfunc(10); // Error! ← Which version of
                                myfunc() should
                                this use?

    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`. However, when `myfunc()` is called 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`. Both are valid conversions. This confusion causes an error message to be displayed and prevents the program from compiling.

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

Here is another example of ambiguity caused by the automatic type conversions in C++:

```
// Another example of overloading ambiguity.


#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) << " "; // Error, ambiguous!
    return 0;
}

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

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



Should 88 be converted to  
char or unsigned char?

In C++, `unsigned char` and `char` are not inherently ambiguous. (They are different types.) However, when `myfunc()` is called with the integer 88, the compiler does not know which function to call. That is, should 88 be converted into a `char` or `unsigned char`? Both are valid conversions.

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

```
// More function overloading ambiguity.
```

```
#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); // Error, ambiguous!
    return 0;
}
```

Here, is **j** defaulting or is the single-parameter version of **myfunc()** being invoked?

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

In the first call to `myfunc()`, two arguments are specified; therefore, no ambiguity is introduced, and `myfunc(int i, int j)` is called. However, the second call to `myfunc()` results in ambiguity, 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.

As you continue to write your own C++ programs, be prepared to encounter ambiguity errors. Unfortunately, until you become more experienced, you will find that they are fairly easy to create.

## Module 6 Mastery Check

1. What are the two ways that an argument can be passed to a subroutine?
2. In C++, what is a reference? How is a reference parameter created?
3. Given this fragment,  

```
int f(char &c, int *i);
// ...
char ch = 'x'; int i = 10;
```

show how to call `f()` with the `ch` and `i`.
4. Create a void function called `round()` that rounds the value of its double argument to the nearest whole value. Have `round()` use a reference parameter and return the rounded result in this parameter. You can assume that all values to be rounded are positive. Demonstrate `round()` in a program. To solve this problem, you will need to use the `modf()` standard library function, which is shown here:  

```
double modf(double num, double *i);
```



The `modf( )` function decomposes `num` into its integer and fractional parts. It returns the fractional portion and places the integer part in the variable pointed to by `i`. It requires the header `<cmath>`.

5. Modify the reference version of `swap( )` so that in addition to exchanging the values of its arguments, it returns a reference to the smaller of its two arguments. Call this function `min_swap( )`.
6. Why can't a function return a reference to a local variable?
7. How must the parameter lists of two overloaded functions differ?
8. In Project 6-1, you created a collection of `print( )` and `println( )` functions. To these functions, add a second parameter that specifies an indentation level. For example, when `print( )` is called like this,  

```
print("test", 18);
```

output will indent 18 spaces and then will display the string "test". Have the indentation parameter default to 0 so that when it is not present, no indentation occurs.
9. Given this prototype,  

```
bool myfunc(char ch, int a=10, int b=20);
```

show the ways that `myfunc( )` can be called.
10. Briefly explain how function overloading can introduce ambiguity.