FUNCTION COMPONENTS AND ARGUMENTS PASSING IN CPP

The General Form of a Function

The general form of a function is

ret-type function-name(*parameter list*)

{

body of the function

}

The *ret-type* specifies the type of data that the function returns.Afunction may return any type of data except an array. The *parameter list* is a comma-separated list of variable names and their associated types that receive the values of the arguments when the function is called.Afunction may be without parameters, in which case the parameter list is empty. However, even if there are no parameters, the parentheses are still required. In variable declarations, you can declare many variables to be of a common type by using a comma-separated list of variable names. In contrast, all function parameters must be declared individually, each including both the type and name. That is, the parameter declaration list for a function takes this general form:

f(type varname1, type varname2, . . . , type varnameN)

For example, here are correct and incorrect function parameter declarations:

f(int i, int k, int j) /* correct */

f(int i, k, float j) /* incorrect */

Scope Rules of Functions

The *scope rules* of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. Each function is a discrete block of code. Afunction's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function. (For instance, you cannot use **goto** to jump into the middle of another function.) The code that constitutes the body of a function is hidden from the rest of the program and, unless it uses global variables or data, it can neither

affect nor be affected by other parts of the program. Stated another way, the code and data that are defined within one function cannot interact with the code or data defined in another function because the two functions have a different scope. Variables that are defined within a function are called *local* variables. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, local variables cannot hold their value between function calls. The only exception to this rule is when the variable is declared with the **static** storage class specifier. This causes the compiler to treat the variable as if it were a global variable for storage purposes, but limits its scope to within the function. In C (and C++) you cannot define a function within a function. This is why neither C nor C++ are technically block-structured languages.

Argument passing

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the *formal parameters* of the function. They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. As shown in the following function, the parameter declarations occur after the function name:

```
/* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s, char c)
{
while(*s)
if(*s==c) return 1;
else s++;
return 0;
}
```

The function **is_in( )** has two parameters: **s** and **c**. This function returns 1 if the character **c** is part of the string **s**; otherwise, it returns 0. As with local variables, you may make assignments to a function's formal parameters or use them in an expression. Even though these variables perform the special task of receiving the value of the arguments passed to the function, you can use them as you do any other local variable.

Call by Value, Call by Reference

In a computer language, there are two ways that arguments can be passed to a subroutine. The first is known as *call by value*. This method copies the *value of* an argument into the formal parameter of the subroutine. In this case, changes made to the parameter have no effect on the argument. *Call by reference* is the second way of passing arguments to a subroutine. In this method, the *address* of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. By default, C/C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

Consider the following program:

```
#include <stdio.h>
int sqr(int x);
int main(void)
{
int t=10;
printf("%d %d", sqr(t), t);
return 0;
}
int sqr(int x)
{
x = x*x;
return(x);
}
```

In this example, the value of the argument to **sqr( )**, 10, is copied into the parameter **x**. When the assignment **x = x*x** takes place, only the local variable **x** is modified. The variable **t**, used to call **sqr( )**, still has the value 10. Hence, the output is **100 10**. Remember that it is a copy of the value of the argument that is passed into the function. What occurs inside the function has no effect on the variable used in the call.

Creating a Call by Reference

Even though C/C++ uses call by value for passing parameters, you can create a call by reference by passing a pointer to an argument, instead of the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function. Pointers are passed to functions just like any other value. Of course, you need to declare the parameters as pointer types.

For example, the function **swap( )**, which exchanges the values of the two integer variables pointed to by its arguments, shows how.

```
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 */
}
```

swap( ) is able to exchange the values of the two variables pointed to by **x** and **y** because their addresses (not their values) are passed. Thus, within the function, the contents of the variables can be accessed using standard pointer operations, and the contents of the variables used to call the function are swapped.

The return Statement

The **return** statement itself is described in Chapter 3. As explained, it has two important uses. First, it causes an immediate exit from the function that it is in. That is, it causes program execution to return to the calling code. Second, it may be used to return a value.

This section examines how the **return** statement is used.