Lab Session 11

Functions in C++ Language (Part-1).

Objectives:

1. Illustration of functions.
2. To learn the syntax and semantics of the Functions in C++ programming language.
3. Demonstrate a thorough understanding of Functions through logic building and implementing programs logic.

Simple Functions:

A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program.

Syntax:

You've already seen how to create a function; every single one of your programs has had a main function in it! Let's take another function, to have something to talk about and really pull apart all the pieces of a function:

```
int add (int x, int y)
{
return x + y;
}
```

Okay, so what's going on? First, notice that this looks a lot like the main function that you've written several times already. There are only two real differences:

1. This function takes two arguments, x and y. Main did not take any arguments.
2. This function explicitly returns a value (remember that main also returns a value, but you don't have to put in the return statement yourself).

The line

```
int add (int x, int y)
```

gives the return type first, before the function name. The two arguments are listed after the name. If you take no arguments, you'd simply write a pair of parentheses, like this:

```
int no_arg_function ()
```

If you want a function that does not return a value—for example, a function that just prints something to the screen—you can declare its return type as **void**. This will prevent you from using your function as an expression (such as in variable assignments or the condition of an if statement).

The return value is provided by using the return statement; this function consists of only a single line,
return x + y;

But you can have more than one line, just like in main, and the function will stop only when the return statement runs, providing the value to the caller.
Once you've declared your function, you can then call your newly-minted function like this:

add(1, 2); // ignore the return value

You can also use the function as an expression to assign it to a variable or output it:

```
#include <iostream>
using namespace std;
int add (int x, int y)
{
        return x + y;
}
int main ()
{
        int result = add( 1, 2 ); // call add and assign the result to a                    //variable
        cout << "The result is: " << result << '\n';
        cout << "Adding 3 and 4 gives us: " << add( 3, 4 );
}
```

In this example, it might look like cout will output the add function. But as with variables, cout prints the result of the expression rather than the literal phrase "add(3, 4)". The result would be the same as if we had run this line of code:

```
        cout << "Adding 3 and 4 gives us: " << 3 + 4;
```

In the example program, notice that we call the add function several times, rather than repeating the code again and again. For such a short function, that doesn't really help us much, but if we

later decide to add some more code to the add function (maybe some debugging statements to print out the arguments and result) it means we'd have to change much less code—just the function, rather than every place that had the duplicated code.

Local variables and global variables

Now that you can have more than one function, you will probably have many more variables, some in
each function. Let's talk for a minute about the names you give variables. When you declare a variable
inside a function, you give it a name. Where can you use that name to refer to that variable?

Local variables

Let's take a simple function:

```
Int
{
        int result = x + 10;
        return result;
}
```

There are two variables here, x and result. Let's talk about result first—the variable result is available only within the curly braces in which it is defined—basically, the two lines within the add function. In other words, you could also write another function with the variable result:

```
int getValueTen ()
{
        int result = 10;
        return result;
}
```

You could even use getValueTen inside addTen

```
int addTen (int x)
        {
                int result = x + getValueTen();
                return result;
}
```

There are two different variables called result, one that belongs to the addTen function and another that belongs to the getValueTen function. The variables do not conflict—while getValueTen executes, it has access only to its own copy of the result variable, and vice-versa.

The visibility of a variable is called its **scope**. The scope of a variable simply means the section of code where the variable's name can be used to access that variable. Variables declared within a function are available only in the scope of the function—when the function itself is executing. Variables declared in the scope of one function are not available to other functions that are called during execution of the first function. When one function calls another, the new function's variables are the only ones available.

Arguments to functions are also declared in the scope of the function. These variables are not available to the caller of the function—even though the caller is providing the value. The variable x, in the addTen function, is an argument to the function, and can only be used inside the addTen function. Morever, like any other variable declared within one function, the variable x cannot be used by function that addTen calls. In the example above, the variable x, an argument to addTen, is not available to the getValueTen function.

Function arguments are like the stunt-doubles of the variables passed in to the function; changing a function argument has no effect on the original variable. To make this happen, when a variable is passed into a function, it is copied into the function argument:

```
#include <iostream>
using namespace std;
void changeArgument (int x)
{
	x = x + 5;
}
int main()
{
	int y = 4;
	changeArgument( y ); // y will be unharmed by the function call
	cout << y; // still prints 4
}
```

The scope of a variable can be even narrower than an entire function. Every set of curly braces defines a new, more narrow scope. For example:

```
int divide (int numerator, int denominator)
{
	if ( 0 == denominator )
	{
		int result = 0;
```

```
            return result;
        }
        int result = numerator / denominator;
        return result;
}
```

The first declaration of result is in scope only within the if statement's curly braces. The second declaration of result is in scope only from the place where it was declared to the end of the function. In general, the compiler won't stop you from creating two variables with the same name, as long as they are used in different scopes. In cases such as in the divide function, multiple variables with the same name in similar scopes can be confusing to someone trying to understand the code.

Any variable declared in the scope of a function, or inside of a block, is called a **local variable**. You can also have variables that are available more widely, called global variables.

Global variables

Sometimes you want to have a single variable that is available to all of your functions. For example, if you have a board game, you might want to store the board as a global variable so that you can have multiple functions that use the board without having to pass it around all the time.

You can accomplish this by using a global variable. A **global variable** is a variable that is declared outside of any function. These variables are available everywhere in the program past the point of the variable's declaration.

Here's a basic example of a global variable showing how you declare it, and how you can use it.

```
#include <iostream>
using namespace std;
int doStuff () // just a small function to demonstrate scope
{
        return 2 + 3;
}
// global variables can be initialized just like other variables
int count_of_function_calls = 0;

void fun ()
{
        // and the global variable is available here count_of_function_calls++;
}
int main ()
```

```
{
	fun();
	fun();
	fun();
	// and the globla varible is also available here!
	cout << "Function fun was called " << count_of_function_calls << "times";
}
```

The variable count_of_function_calls begins its scope right before the function fun. The function doStuff does not have access to the variable because the variable was declared after doStuff, and both fun and main do have access because they were declared after the variable.

A warning about global variables

Global variables might seem like they make things easier, because everyone can use them. But using global variables makes your code more difficult to understand: to know how a global variable is really used, you have to look everywhere! Using a global variable is rarely the right thing to do. You should use them only when you truly need something to be very widely available. Prefer passing arguments to functions, rather than having functions access global variables. Even when you think that a particular thing is going to be globally used, it may turn out later that it isn't.

Take the game board example from earlier—you might decide to create a function to display the board and have that function access a global variable. But what happens if you want to display some board other than the current board—for example, to show an alternative move? Your function doesn't take the board as an argument; it shows only the single global board. Not very convenient!

Making functions available for use

The rules of scoping that apply to variables—such as a variable being usable only after it is declared— also appy to functions. (Isn't consistency great?)

For example, this program would not compile:

BAD CODE

```
#include <iostream> // needed for cout
using namespace std;
int main ()
{
	int result = add( 1, 2 );
	cout << "The result is: " << result << '\n';
	cout << "Adding 3 and 4 gives us: " << add( 3, 4 );
```

```
}
int add (int x, int y)
{
        return x + y;
}
```

If you compile this program, you will see this error message (or something like it):

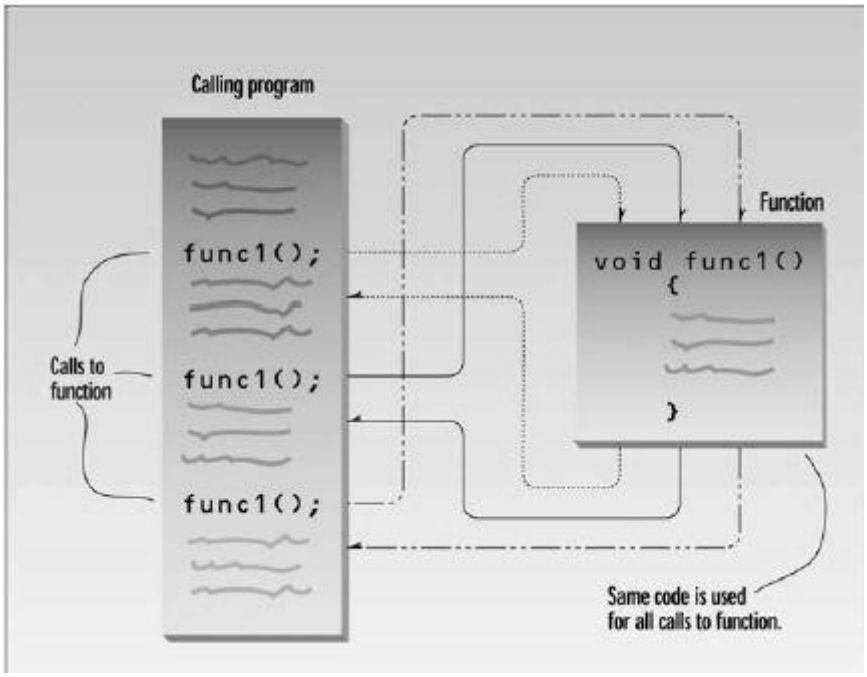
```
        badcode.cpp:7: error: 'add' was not declared in this scope
```

The problem is that at the point where the add function is called, it hasn’t been declared yet, so it was not in scope. When the compiler sees you try to call a function you haven't declared, it gets very confused—poor compiler!

One solution, which I used in earlier examples, is just to put the whole function above the places that use it. Another solution is to **declare** the function before you **define** it.

Although declaring a function and defining a function sound very similar, they have very different meanings, so let’s break down the terminology.

Another reason to use functions (and the reason they were invented, long ago) is to reduce program size. Any sequence of instructions that appears in a program more than once is a candidate for being made into a function. The function’s code is stored in only one place in memory, even though the function is executed many times in the course of the program. Figure 1 shows how a function is invoked from different sections of a program.



Our first example demonstrates a simple function whose purpose is to print a line of 45 asterisks. The example program generates a table, and lines of asterisks are used to make the table more readable. Here's the listing for TABLE:

```
// table.cpp
// demonstrates simple function
#include <iostream>
using namespace std;

void starline(); //function declaration  (prototype)

int main()
        {
        starline(); //call to function
        cout << "Data type Range" << endl;
        starline(); //call to function
        cout << "char -128 to 127" << endl
        << "short -32,768 to 32,767" << endl
        << "int System dependent" << endl
        << "long -2,147,483,648 to 2,147,483,647" << endl;
        starline(); //call to function
        return 0;
        }

//-------------------------------------------------------------
// starline()
// function definition
        void starline()              //function declarator
```

```
{
for(int j=0; j<45; j++)     //function body
cout << '*';
cout << endl;
}
```

The output from the program looks like this:

```
*********************************************
Data type Range
*********************************************
char -128 to 127
short -32,768 to 32,767
int System dependent
long -2,147,483,648 to 2,147,483,647
*********************************************
```

The program consists of two functions: main() and starline(). You've already seen many programs that use main() alone. What other components are necessary to add a function to the program? There are three: the function *declaration*, the *calls* to the function, and the function *definition*.

The Function Declaration:

Just as you can't use a variable without first telling the compiler what it is, you also can't use a function without telling the compiler about it. There are two ways to do this. The approach we show here is to *declare* the function before it is called. (The other approach is to *define* it before it's called; we'll examine that next.) In the TABLE program, the function starline() is declared in the line.

```
void starline();
```

The declaration tells the compiler that at some later point we plan to present a function called *starline*. The keyword void specifies that the function has no return value, and the empty parentheses indicate that it takes no arguments. (You can also use the keyword void in parentheses to indicate that the function takes no arguments, as is often done in C, but leaving them empty is the more common practice in C++.) We'll have more to say about arguments and return values soon.

Notice that the function declaration is terminated with a semicolon. It is a complete statement in itself. Function declarations are also called *prototypes*, since they provide a model or blueprint for the function. They tell the compiler, "a function that looks like this is coming up later in the program, so it's all right if you see references to it before you see the function itself." The information in the declaration (the return type and the number and types of any arguments) is also sometimes referred to as the function *signature*.

Calling the Function:

The function is *called* (or *invoked*, or *executed*) three times from main(). Each of the three calls looks like this:

```
starline();
```

This is all we need to call the function: the function name, followed by parentheses. The syntax of the call is very similar to that of the declaration, except that the return type is not used. The call is terminated by a

semicolon. Executing the call statement causes the function to execute; that is, control is transferred to the function, the statements in the function definition (which we'll examine in a moment) are executed, and then control returns to the statement following the function call.

The Function Definition:

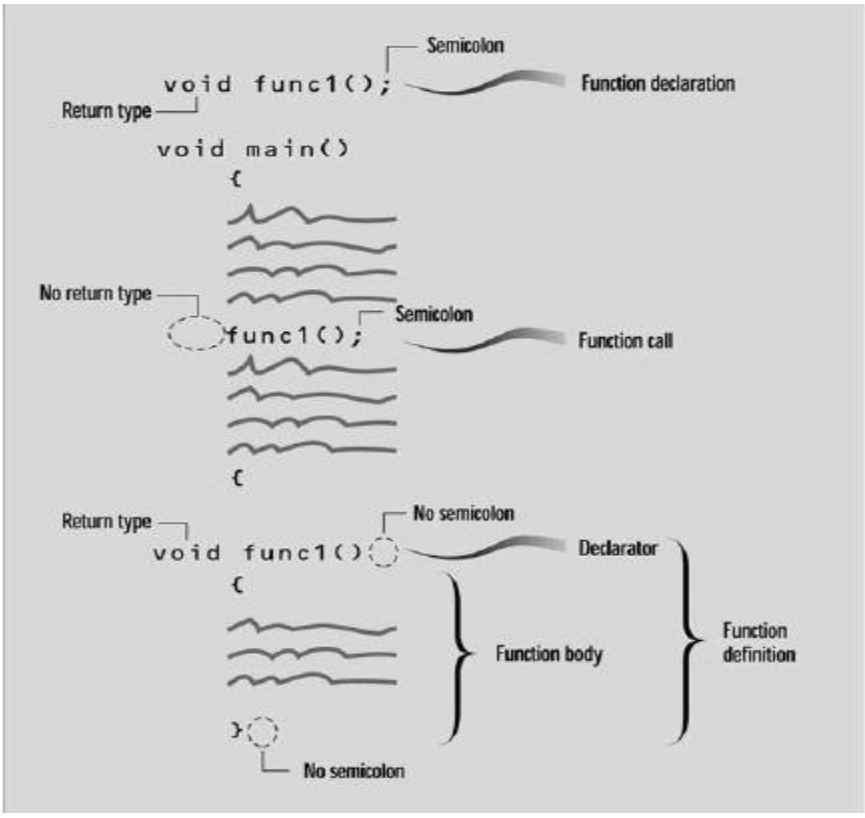
Finally we come to the function itself, which is referred to as the function *definition*. The definition contains the actual code for the function. Here's the definition for starline():

```
void starline() //declarator
        {
        for(int j=0; j<45; j++) //function body
        cout << '*';
        cout << endl;
        }
```

The definition consists of a line called the *declarator*, followed by the function *body*. The function body is composed of the statements that make up the function, delimited by braces.

The declarator must agree with the declaration: It must use the same function name, have the same argument types in the same order (if there are arguments), and have the same return type.

Notice that the declarator is *not* terminated by a semicolon. Figure 2 shows the syntax of the function declaration, function call, and function definition.



When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed, and when the closing brace is encountered, control returns to the calling program.

Table 1 summarizes the different function components.

Component	*Purpose*	*Example*
Declaration (prototype)	Specifies function name, argument types, and return value. Alerts compiler (and programmer) that a function is coming up later.	`void func();`
Call	Causes the function to be executed.	`func();`

Definition	The function itself. Contains the lines of code that constitute the function.	`void func()`
		`{`
		`// lines of code`
		`}`

Declarator	First line of definition.	`void func()`

Comparison with Library Functions:

We've already seen some library functions in use. We have embedded calls to library functions, such as
ch = getche();

In our program code. Where are the declaration and definition for this library function? The declaration is in the header file specified at the beginning of the program (CONIO.H, for getche()). The definition (compiled into executable code) is in a library file that's linked automatically to your program when you build it. When we use a library function we don't need to write the declaration or definition. But when we write our own functions, the declaration and definition are part of our source file, as we've shown in the TABLE example.

Eliminating the Declaration:

The second approach to inserting a function into a program is to eliminate the function declaration and place the function definition (the function itself) in the listing before the first call to the function. For example, we could rewrite TABLE to produce TABLE2, in which the definition for starline() appears first.

```
// table2.cpp
// demonstrates function definition preceding function calls
#include <iostream>
using namespace std; //no function declaration
//--------------------------------------------------------------
// starline() //function definition
        void starline()
```

```
{
for(int j=0; j<45; j++)
cout << '*';
cout << endl;
}
//--------------------------------------------------------------
int main() //main() follows function
{
starline(); //call to function
cout << "Data type Range" << endl;
starline(); //call to function
cout << "char -128 to 127" << endl
<< "short -32,768 to 32,767" << endl
<< "int System dependent" << endl
<< "long -2,147,483,648 to 2,147,483,647" << endl;
starline(); //call to function
return 0;
}
```

This approach is simpler for short programs, in that it removes the declaration, but it is less flexible. To use this technique when there are more than a few functions, the programmer must give considerable thought to arranging the functions so that each one appears before it is called by any other. Sometimes this is impossible. Also, many programmers prefer to place main() first in the listing, since it is where execution begins. In general we'll stick with the first approach, using declarations and starting the listing with main().

Passing Arguments to Functions:

An *argument* is a piece of data (an int value, for example) passed from a program to the function. Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the program calling it.

Passing Constants

As an example, let's suppose we decide that the starline() function in the last example is too rigid. Instead of a function that always prints 45 asterisks, we want a function that will print any character any number of times.

Here's a program, TABLEARG, that incorporates just such a function. We use arguments to pass the character to be printed and the number of times to print it.

```
// tablearg.cpp
// demonstrates function arguments
#include <iostream>
using namespace std;
void repchar(char, int); //function declaration
        int main()
        {
        repchar('-', 43); //call to function
        cout << "Data type Range" << endl;
        repchar('=', 23); //call to function
        cout << "char -128 to 127" << endl
```

```
        << "short -32,768 to 32,767" << endl
        << "int System dependent" << endl
        << "double -2,147,483,648 to 2,147,483,647" << endl;
        repchar('-', 43); //call to function
        return 0;
        }
//--------------------------------------------------------------
// repchar()
// function definition
void repchar(char ch, int n) //function declarator
        {
        for(int j=0; j<n; j++) //function body
        cout << ch;
        cout << endl;
        }
```

The new function is called repchar(). Its declaration looks like this: void repchar(char, int); // declaration specifies data types The items in the parentheses are the data types of the arguments that will be sent to repchar():
char and int.
In a function call, specific values—constants in this case—are inserted in the appropriate place in the parentheses:
repchar('-', 43); // function call specifies actual values
This statement instructs repchar() to print a line of 43 dashes. The values supplied in the call must be of the types specified in the declaration: the first argument, the - character, must be of type char; and the second argument, the number 43, must be of type int. The types in the declaration and the definition must also agree.

The next call to repchar()
repchar('=', 23);
tells it to print a line of 23 equal signs. The third call again prints 43 dashes. Here's the output from TABLEARG:

```
-------------------------------------------
Data type Range
=======================
char -128 to 127
short -32,768 to 32,767
int System dependent
long -2,147,483,648 to 2,147,483,647
-------------------------------------------
```

The calling program supplies *arguments*, such as '–' and 43, to the function. The variables used within the function to hold the argument values are called *parameters*; in repchar() they are ch and n. (We should note that many programmers use the terms argument and parameter somewhat interchangeably.) The declarator in the function definition specifies both the data types and the names of the parameters:

```
void repchar(char ch, int n) //declarator specifies parameter
```

//names and data types

These parameter names, ch and n, are used in the function as if they were normal variables. Placing them in the declarator is equivalent to defining them with statements like

```
char ch;
int n;
```

When the function is called, its parameters are automatically initialized to the values passed by the calling program.

Passing Variables

In the TABLEARG example the arguments were constants: '–', 43, and so on. Let's look at an example where variables, instead of constants, are passed as arguments. This program, VARARG, incorporates the same repchar() function as did TABLEARG, but lets the user specify the character and the number of times it should be repeated.

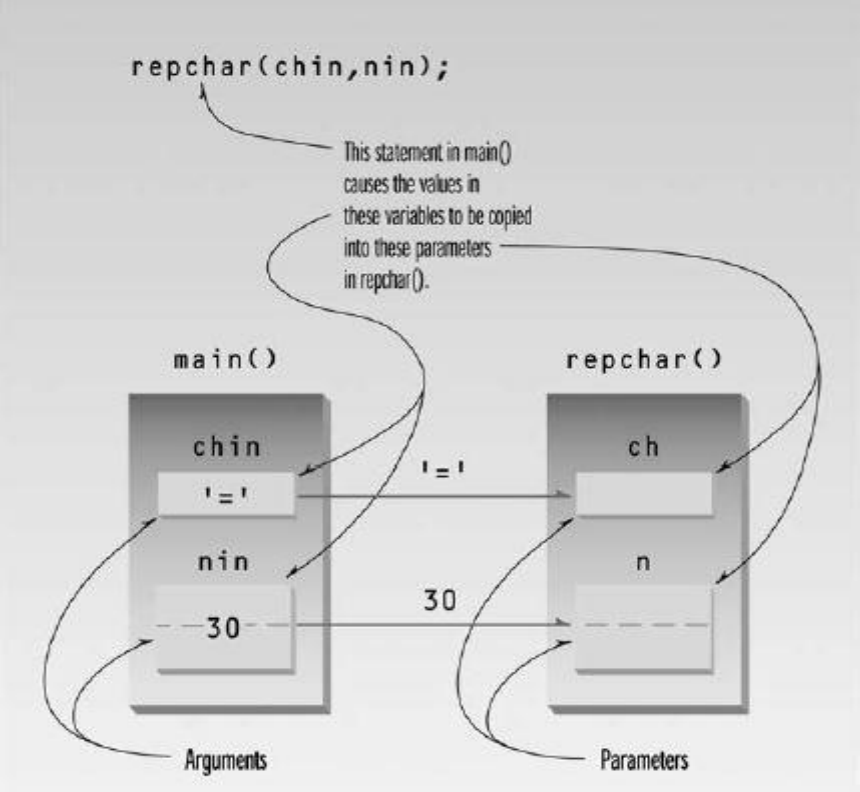
```
// demonstrates variable arguments
#include <iostream>
using namespace std;
void repchar(char, int); //function declaration
int main()
        {
        char chin;
        int nin;
        cout << "Enter a character: ";
        cin >> chin;
        cout << "Enter number of times to repeat it: ";
        cin >> nin;
        repchar(chin, nin);
        return 0;
        }
//-------------------------------------------------------------
// repchar()
// function definition
void repchar(char ch, int n) //function declarator
        {
        for(int j=0; j<n; j++) //function body
        cout << ch;
        cout << endl;
        }
```

Here's some sample interaction with VARARG:
Enter a character: +
Enter number of times to repeat it: 20
++++++++++++++++++++
Here chin and nin in main() are used as arguments to repchar():
repchar(chin, nin); // function call
The data types of variables used as arguments must match those specified in the function declaration and definition, just as they must for constants. That is, ch*in* must be a char, and n*in* must be an int.

Passing by Value

In above program the particular values possessed by chin and nin when the function call is executed will be passed to the function. As it did when constants were passed to it, the function creates new variables to hold the values of these variable arguments. The function gives these new variables the names and data types of the parameters specified in the declarator: ch of type char and n of type int. It initializes these parameters to the values passed. They are then accessed like other variables by statements in the function body.

Passing arguments in this way, where the function creates copies of the arguments passed to it is called passing by value. We'll explore another approach, passing by reference, later in this chapter. Following figure shows how new variables are created in the function when arguments are passed by value.



Passing by value.

Structures as Arguments

Entire structures can be passed as arguments to functions. We'll show two examples, one with the Distance structure, and one with a structure representing a graphics shape.

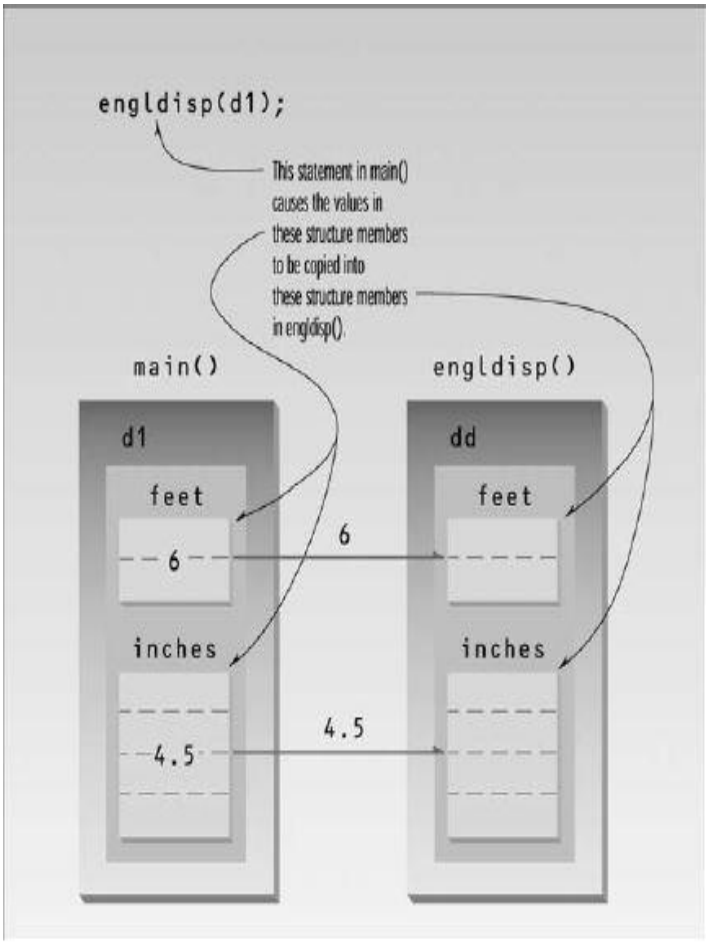
Passing a Distance Structure

This example features a function that uses an argument of type Distance, the same structure type

```
// demonstrates passing structure as argument
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
struct Distance //English distance
        {
        int feet;
        float inches;
        };
////////////////////////////////////////////////////////////

void engldisp( Distance ); //declaration
int main()
```

```
        {
        Distance d1, d2; //define two
        lengths
        //get length d1 from user
        cout << "Enter feet: "; cin >>
        d1.feet;
        cout << "Enter inches: "; cin >>
        d1.inches;
        //get length d2 from user
        cout << "\nEnter feet: "; cin >>
        d2.feet;
        cout << "Enter inches: "; cin >>
        d2.inches;
        cout << "\nd1 = ";
        engldisp(d1); //display length 1
        cout << "\nd2 = ";
        engldisp(d2); //display length 2
        cout << endl;
        return 0;
        }
//--------------------------------------------
--------------
// engldisp()
// display structure of type Distance in feet and inches
void engldisp( Distance dd ) //parameter dd of type Distance
        {
        cout << dd.feet << "\'-" << dd.inches << "\"";
        }
```



The main() part of this program accepts two distances in feet-and-inches format from the user, and places these values in two structures, d1 and d2. It then calls a function, engldisp(), that takes a Distance structure variable as an argument. The purpose of the function is to display the distance passed to it in the standard format, such as 10'–2.25". Here's some sample interaction with the program:

```
Enter feet: 6
Enter inches: 4
Enter feet: 5
Enter inches: 4.25
d1 = 6'-4"
d2 = 5'-4.25"
```

The function declaration and the function calls in main(), and the declarator in the function body, treat the structure variables just as they would any other variable used as an argument; this one just happens to be type Distance, rather than a basic type like char or int.

In main() there are two calls to the function engldisp(). The first passes the structure d1; the second passes d2. The function engldisp() uses a parameter that is a structure of type Distance, which it names dd. As with simple variables, this structure variable is automatically initialized to the value of the structure passed from main(). Statements in engldisp() can then access the members of dd in the usual way, with the expressions dd.feet and dd.inches. Figure shows a structure being passed as an argument to a function. *Structure passed as an argument*

As with simple variables, the structure parameter dd in engldisp() is not the same as the arguments passed to it (d1 and d2). Thus, engldisp() could (although it doesn't do so here) modify dd without affecting d1 and d2. That is, if engldisp() contained statements like

dd.feet = 2;
dd.inches = 3.25;
this would have no effect on d1 or d2 in main().

Names in the Declaration

Here's a way to increase the clarity of your function declarations. The idea is to insert meaningful names in the declaration, along with the data types. For example, suppose you were using a function that displayed a point on the screen. You could use a declaration with only data types

void display_point(int, int); //declaration
but a better approach is
void display_point(int horiz, int vert); //declaration

These two declarations mean exactly the same thing to the compiler. However, the first approach, with (int, int), doesn't contain any hint about which argument is for the vertical coordinate and which is for the horizontal coordinate. The advantage of the second approach is clarity for the programmer: Anyone seeing this declaration is more likely to use the correct arguments when calling the function.

Note that the names in the declaration have no effect on the names you use when calling the function. You are perfectly free to use any argument names you want:
display_point(x, y); // function call
We'll use this name-plus-datatype approach when it seems to make the listing clearer.

Returning Values from Functions

When a function completes its execution, it can return a single value to the calling program. Usually this return value consists of an answer to the problem the function has solved. The next example demonstrates a function that returns a weight in kilograms after being given a weight in pounds. Here's the listing for CONVERT:

```
//CONVERT
// demonstrates return values, converts pounds to kg
#include <iostream>
using namespace std;
float lbstokg(float); //declaration
int main()
	{
	float lbs, kgs;
	cout << "\nEnter your weight in pounds: ";
	cin >> lbs;
	kgs = lbstokg(lbs);
	cout << "Your weight in kilograms is " << kgs << endl;
	return 0;
	}

//-------------------------------------------------------------
// lbstokg()
// converts pounds to kilograms
float lbstokg(float pounds)
	{
	float kilograms = 0.453592 * pounds;
	return kilograms;
	}
```

Here's some sample interaction with this program:

```
Enter your weight in pounds: 182
Your weight in kilograms is 82.553741
```

When a function returns a value, the data type of this value must be specified. The function declaration does this by placing the data type, float in this case, before the function name in the declaration and the definition. Functions in earlier program examples returned no value, so the return type was void. In the CONVERT program, the function lbstokg() (*pounds to kilograms*, where lbs means pounds) returns type float, so the declaration is

```
float lbstokg(float);
```

The first float specifies the return type. The float in parentheses specifies that an argument to be passed to lbstokg() is also of type float.

When a function returns a value, the call to the function

lbstokg(lbs)

is considered to be an expression that takes on the value returned by the function. We can treat this expression like any other variable; in this case we use it in an assignment statement:

kgs = lbstokg(lbs);

This causes the variable kgs to be assigned the value returned by lbstokg().

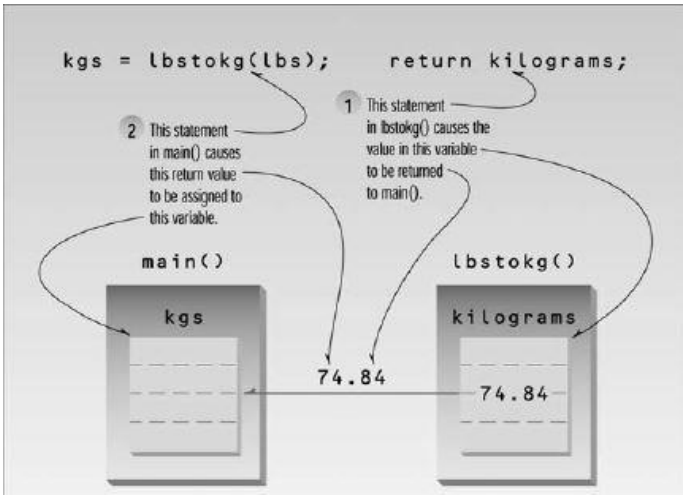
The return Statement

The function lbstokg() is passed an argument representing a weight in pounds, which it stores in the parameter pounds. It calculates the corresponding weight in kilograms by multiplying this pounds value by a constant; the result is stored in the variable kilograms. The value of this variable is then returned to the calling program using a return statement:

return kilograms;

Notice that both main() and lbstokg() have a place to store the kilogram variable: kgs in main(), and kilograms in lbstokg(). *Returning a Value*

When the function returns, the value in kilograms is *copied into* kgs. The calling program does not access the kilograms variable in the function; only the value is returned. This process is shown in Figure below.



While many arguments may be sent to a function, only one argument may be returned from it. This is a limitation when you need to return more information. However, there are other approaches to returning multiple variables from functions. One is to pass arguments by reference, which we'll look at later in this chapter. Another is to return a structure with the multiple values as members, as we'll see soon.

You should always include a function's return type in the function declaration. If the function doesn't return anything, use the keyword void to indicate this fact. If you don't use a return type in the declaration, the compiler will assume that the function returns an int value. For example, the declaration

somefunc(); // declaration -- assumes return type is int tells the compiler that somefunc() has a return type of int.

The reason for this is historical, based on usage in early versions of C. In practice, you shouldn't take advantage of this default type. Always specify the return type explicitly, even if it actually is int. This keeps the listing consistent and readable.

Eliminating Unnecessary Variables

The following program contains several variables that are used in the interest of clarity but are not really necessary. A variation of this program, CONVERT2, shows how expressions containing functions can often be used in place of variables.

```
//CONVERT2
// eliminates unnecessary variables
#include <iostream>
using namespace std;
float lbstokg(float); //declaration
int main()
        {
        float lbs;
        cout << "\nEnter your weight in pounds: ";
        cin >> lbs;
        cout << "Your weight in kilograms is " << lbstokg(lbs)
        << endl;
        return 0;
        }

//-------------------------------------------------------------
// lbstokg()
// converts pounds to kilograms
float lbstokg(float pounds)
        {
        return 0.453592 * pounds;
        }
```

In main() the variable kgs from the CONVERT program has been eliminated. Instead the function lbstokg(lbs) is inserted directly into the cout statement:

cout << "Your weight in kilograms is " << lbstokg(lbs) << endl;

Also in the lbstokg() function, the variable kilograms is no longer used. The expression 0.453592*pounds is inserted directly into the return statement:

return 0.453592 * pounds;

The calculation is carried out and the resulting value is returned to the calling program, just as the value of a variable would be.

For clarity, programmers often put parentheses around the expression used in a return statement:

return (0.453592 * pounds);

Even when not required by the compiler, extra parentheses in an expression don't do any harm, and they may help make the listing easier for us poor humans to read.

Experienced C++ (and C) programmers will probably prefer the concise form of CONVERT2 to the more verbose CONVERT. However, CONVERT2 is not so easy to understand, especially for the non-expert. The brevity-versus-clarity issue is a question of style, depending on your personal preference and on the expectations of those who will be reading your code.

Quiz yourself

1. Which is not a proper prototype?
    A. int funct(char x, char y);
    B. double funct(char x)
    C. void funct();
    D. char x();
2. What is the return type of the function with prototype: int func(char x, double v, float t);
    A. char
    B. int
    C. float

D. double

3. Which of the following is a valid function call (assuming the function exists)?
    A. funct;
    B. funct x, y;
    C. funct();
    D. int funct();
4. Which of the following is a complete function?
    A. int funct();
    B. int funct(int x) {return x=x+1;}
    C. void funct(int) {cout<<"Hello"}
    D. void funct(x) {cout<<"Hello";}

Task:

1. Make your calculator program perform computations in a separate function for each type of computation.
2. Modify your password program from before to put all of the password checking logic into a separate function, apart from the rest of the program.