

Computer Programming [Lab]

Lab#03: Functions

Agenda

- Library Functions
- User Defined Functions
 - Value Returning Functions
 - Void Functions
- Why define your own Functions
- Functions Prototype/Declaration
- Scope of Variables
- Parameter Types
 - Formal Parameters
 - Actual Parameters
- Calling Functions
 - Pass by Value
 - Pass by Reference
- Comparison between Pass by Value/Reference
- Default Arguments

In programming, function refers to a segment that group's code to perform a specific task. Depending on whether a function is predefined or created by programmer; there are two types of function:

- Library Function
- User-defined Function

C++ Predefined Functions (Library Functions)

Library functions are the built-in function in C++ programming. Programmer can use library function by invoking function directly; they don't need to write it themselves.

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

int main()
{
    double number, squareRoot;
    cout << "Enter a number: ";
    cin >> number;

    // sqrt() is a library function to calculate square root
    squareRoot = sqrt(number);
    cout << "Square root of " << number << " = " << squareRoot;
    return 0;
}
```

Enter a number: 26

Square root of 26 =
5.09902

In the example above, `sqrt()` library function is invoked to calculate the square root of a number.

Notice code `#include <cmath>` in the above program. Here, `cmath` is a header file. The function definition of `sqrt()` (body of that function) is present in the `cmath` header file.

You can use all functions defined in `cmath` when you include the content of file `cmath` in this program using `#include <cmath>`.

Every valid C++ program has at least one function, that is, `main()` function.

Types of User-defined Functions

User-defined functions in C++ are classified into two categories:

- Value returning functions
- Void functions

Value-returning functions

C++ allows programmer to define their own function.

A function is a group of statements to perform a specific task. Function is given a name, and which can be called from some point of the program.

Performing one task – E.g. Compute the n!, calculate square root of a number etc.

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
    Body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **No. of Parameters:** A parameter is so called an argument. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter. Parameters are optional; that is, a function may contain no parameters.
- **Data Type of Each Parameter:** The parameter list refers to the type, order, and number of the parameters of a function. So you've to mention each parameter along its data-type.
- **Return Type:** A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the **return_type** is the keyword **void**.
- **Function Body:** The function body contains a collection of statements that define what the function does. The first four properties form what is called the **heading** of the function (also called **the function header**); the fifth property is called the **body** of the function. Together, these five properties form what is called the **definition** of the function.

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

int addition (int a, int b)
{
    int r;
    r=a+b;
    return r;
}
int main ()
{
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
}
```

The result is 8

Why define your own functions?

- Readability: `sqrt(5)` is clearer than copy-pasting in an algorithm to compute the square root
- Maintainability: To change the algorithm, just change the function (vs changing it everywhere you ever used it)
- Code reuse: Lets other people use functions/algorithms you've implemented

Use of Value-returning Function

This suggests that a value-returning function is used:

- In an assignment statement.
- As a parameter in a function call.
- In an output statement.

Return Statement:

- Returns data, and control goes to function's caller, If no data to return, use only `return`;
- Function ends when reaches right brace or `return`
- Functions cannot be defined inside other functions

When a `return` statement executes in a function, the function immediately terminates and the control goes back to the caller. Moreover, the function call statement is replaced by the value returned by the `return` statement. When a `return` statement executes in the function main, the program terminates

```
// function example-1
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
    int r;
    r=a-b;
    return r;
}

int main ()
{
    int x=5, y=3, z;
    z = subtraction (7,2);
    cout << "The first result is " << z << '\n';
    cout << "The second result is " << subtraction (7,2) << '\n';
    cout << "The third result is " << subtraction (x,y) << '\n';
    z= 4 + subtraction (x,y);
    cout << "The fourth result is " << z << '\n';
    system("pause");
}
```

```
// Alternative way
int subtraction (int a, int b)
{
    return a-b;
}
```

Output

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

More Examples...

```
int absolute (int x)
{
    if ( x<0 )
        x = -x;

    return x;
}
```

```
int max (int a, int b)
{
    if ( a>b )
        return a;
    else
        return b;
}
```

```
int factorial (int x)
{
    int ans = 1;
    for(int i=x; i>0; i--)
        ans = ans*i;

    return ans;
}
```

The return value of main

You may have noticed that the return type of `main` is `int`, but most examples in this and earlier chapters did not actually return any value from `main`.

Well, there is a catch: If the execution of `main` ends normally without encountering a `return` statement the compiler assumes the function ends with an implicit return statement:

```
return 0;
```

Note that this only applies to function `main` for historical reasons. All other functions with a `return` type shall end with a proper `return` statement that includes a return value, even if this is never used.

When `main` returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. Other values may be returned by `main`, and some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms. The values for `main` that are guaranteed to be interpreted in the same way on all platforms are:

Value	Description
0	The program was successful
EXIT_SUCCESS	The program was successful (same as above). This value is defined in header <code><cstdlib></code> .
EXIT_FAILURE	The program failed. This value is defined in header <code><cstdlib></code> .

Because the implicit `return 0;` statement for `main` is a tricky exception, some authors consider it good practice to explicitly write the statement.

Function Prototype:

The function heading without the body of the function. We place function prototypes before any function definition (including the definition of main). Function prototype is also known as function declaration.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function sum(), following is the function prototype/declaration:

```
int sum(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

Scope of Variables

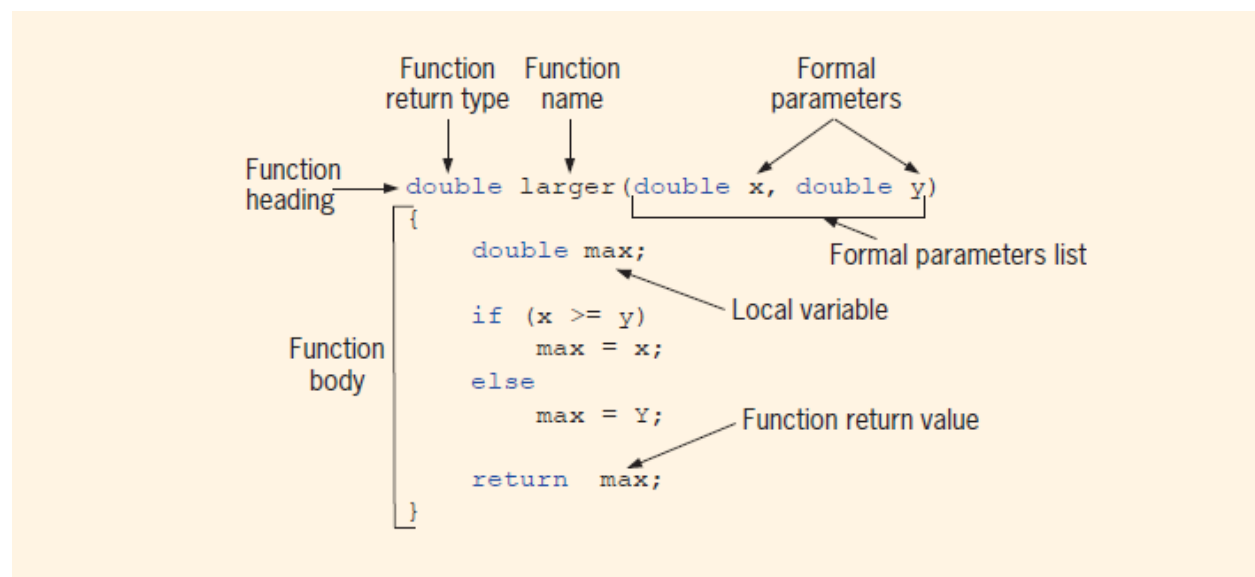


FIGURE 6-1 Various parts of the function `larger`

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables `a`, `b` or `r` directly in function `main` since they were variables

local to function addition. Also, it would have been impossible to use the variable z directly within function addition, since this was a variable local to the function main.

Therefore, the scope of local variables is limited to the same block level in which they are declared.

Nevertheless, we also have the possibility to declare global variables; these are visible from any point of the code, inside and outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

Parameters/Arguments and their Types

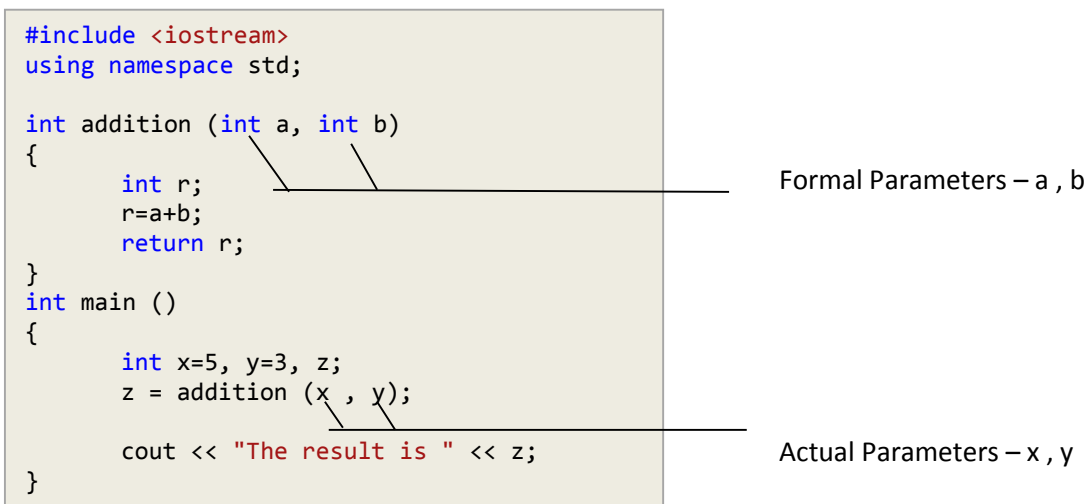
Parameters are the values passed to a function for its working.

There are two types of parameters:

Actual parameters: Parameters that appears in the calling statement of the function are known as actual parameters.

Formal Parameters: Parameters that appears in the function prototype or header statement of the function definition are known as formal parameters. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

Example:



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

int addition (int a, int b)
{
    int r;
    r=a+b;
    return r;
}

int main ()
{
    int x=5, y=3, z;
    z = addition (x , y);
    cout << "The result is " << z;
}
```

The diagram illustrates the distinction between formal and actual parameters in the provided C++ code. The function definition `int addition (int a, int b)` is shown with lines pointing from the text "Formal Parameters – a , b" to the parameters `a` and `b`. The function call `z = addition (x , y);` in the `main` function is shown with lines pointing from the text "Actual Parameters – x , y" to the arguments `x` and `y`.

Calling Functions

There are two ways to call the function by passing the parameters or arguments to a function:

- Call by value / Pass by value
- Call by reference / Pass by reference

Parameters Pass by Value

Until now, in all the functions we have seen, the arguments passed to the functions have been passed by value.

When we call a function by passing parameters by value, the values of actual parameters get copied into the formal parameters but not the variables themselves. The changes made in the values of formal parameters will not be reflected back to the actual parameters. For example, suppose that we called our first function `sum` using the following code:

```
int x=5, y=3, z;
z = sum ( x , y );
```

What we did in this case was to call the function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.

```
int sum ( int a, int b );
z = sum ( 5 , 3 );
```

This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

Parameters Pass by Reference

There might be some cases where we need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
    a*=2;
    b*=2;
    c*=2;
}

int main ()
{
    int x=1, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << ", y=" << y << ", z=" << z;
    return 0;
}
```

```
// OutPut
x=2, y=6, z=14
```


The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed by reference instead of by value.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the formal parameters will have an effect in actual parameter passed as arguments in the call to the function.

```
void duplicate (int& a,int& b,int& c)
               ↑x      ↑y      ↑z
duplicate (   x   ,   y   ,   z   );
```

To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on b will affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

We had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and z without having been modified.

```
// passing parameters by value
#include <iostream>
using namespace std;

void duplicate (int a, int b, int c)
{
    a*=2;
    b*=2;
    c*=2;
    cout << "a=" << a << ", b=" << b << ", c=" << c;
}

int main ()
{
    int x=1, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << ", y=" << y << ", z=" << z;
    return 0;
}
```

```
// Output
```

```
a=2, b=6, c=14
x=1, y=3, z=7
```

Note: Passing by reference is also an effective way to allow a function to return more than one value.

A Comparison between:

S.N	Parameters Passed by Value	Parameters Passed by Reference
1	When we call a function by passing parameters by value, the values of actual parameters get copied into the formal parameters but not the variables themselves.	When we call a function by passing parameters by reference, formal parameters create a reference or pointer directly to the actual parameters.
2	The changes made in the values of formal parameters will not be reflected back to the actual parameters.	The changes made in the values of formal parameters will be reflected back to the actual parameters.
3	<p>Example -</p> <pre>// passing parameters by value #include <iostream> using namespace std; void swap (int a, int b) { a=a+b; b=a-b; c=a-b; cout << "Values inside function \n"; cout<<"a=" << a << ", b=" << b; } int main () { int x=1, y=3; swap (x, y); cout << "Values inside main \n"; cout << "x=" << x << ", y=" << y; return 0; } (Note: Changes made in formal parameters (a and b) are not reflected back to actual parameters (x and y))</pre>	<p>Example -</p> <pre>// passing parameters by reference #include <iostream> using namespace std; void swap (int &a, int &b) { a=a+b; b=a-b; c=a-b; cout << "Values inside function \n"; cout<<"a=" << a << ", b=" << b; } int main () { int x=1, y=3; swap (x, y); cout << "Values inside main \n"; cout << "x=" << x << ", y=" << y; return 0; } (Note: Changes made in formal parameters (a and b) are reflected back to actual parameters (x and y))</pre>

Notes on passing arguments

- The numbers of actual arguments and formals argument should be the same. (Exception: Function Overloading)
- The type of first actual argument should match the type of first formal argument. Similarly, type of second actual argument should match the type of second formal argument and so on.
- You may call function a without passing any argument. The number(s) of argument passed to a function depends on how programmer want to solve the problem.
- You may assign default values to the argument. These arguments are known as default arguments.

Default Values for Parameters

When you define a function you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified this default value is ignored and the passed value is used instead.

Default value/s are passed to argument/s in the function prototype.

Consider the following example:

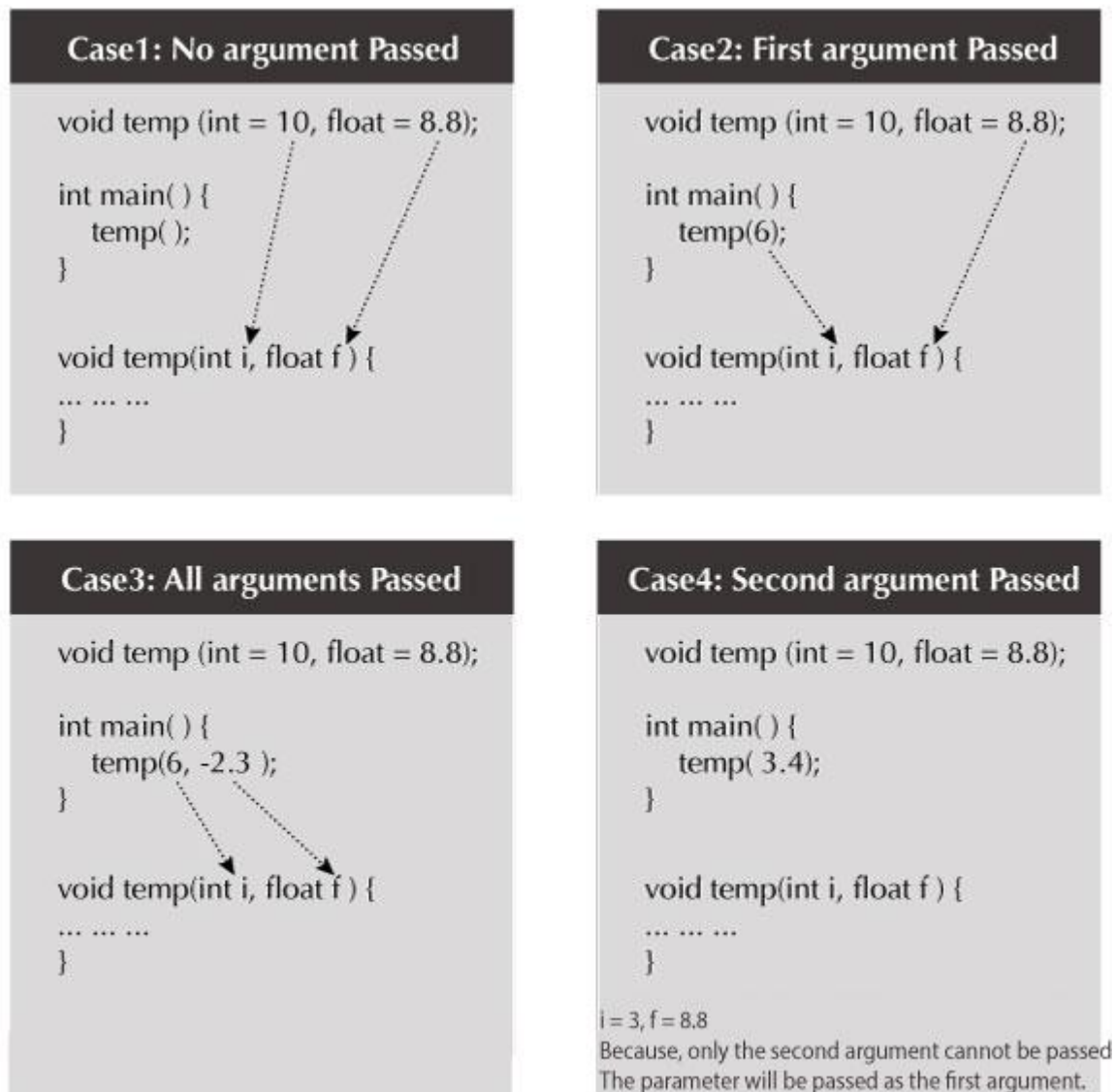


Figure: Working of Default Argument in C++

```
// C++ Program to demonstrate working of default argument

#include <iostream>
using namespace std;

void display(char = '*', int = 1);

int main()
{
    cout << "No argument passed:\n";
    display();

    cout << "\nFirst argument passed:\n";
    display('#');

    cout << "\nBoth argument passed:\n";
    display('$', 5);

    return 0;
}

void display(char c, int n)
{
    for(int i = 1; i <= n; ++i)
    {
        cout << c;
    }
    cout << endl;
}
```

```
// Output

No argument passed:

*

First argument passed:

#

Both argument passed:

$$$$
```

In the above program, you can see the default value assigned to the arguments
`void display(char = '*', int = 1);`.

At first, `display()` function is called without passing any arguments. In this case, `display()` function used both default arguments `c = *` and `n = 1`.

Then, only the first argument is passed using the function second time. In this case, function does not use first default value passed. It uses the actual parameter passed as the first argument `c = #` and takes default value `n = 1` as its second argument.

When `display()` is invoked for the third time passing both arguments, default arguments are not used. So, the value of `c = $` and `n = 5`.

Common mistakes when using Default argument

1. `void add(int a, int b = 3, int c, int d = 4);`

The above function will not compile. You cannot miss a default argument in between two arguments.

In this case, `c` should also be assigned a default value.

2. `void add(int a, int b = 3, int c, int d);`

The above function will not compile as well. You must provide default values for each argument after `b`.

In this case, `c` and `d` should also be assigned default values.

If you want a single default argument, make sure the argument is the last one.

```
void add(int a, int b, int c, int d = 4);
```

3. No matter how you use default arguments, a function should always be written so that it serves only one purpose.

If your function does more than one thing or the logic seems too complicated, you can use Function overloading to separate the logic better.