C++ Programming Language

Presented By:

T.A. Asmaa Hamad El-saied

E-mail: eng.asmaa134@gmail.com

Contents

Functions

- A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.
- You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

Functions(cont.)

- The **C++** standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

- for common mathematical calculations we include the file **math** with the **#include <cmath>** directive which contains the *function prototypes* for the mathematical functions in the **math** library
- Other **header files** which contain the function rand() in <**stdlib** >

Functions(cont.)

- each of which requires the math header file. These functions take one or more double arguments and return a double value.

Function	Description
sqrt(x)	square root
sin(x)	trigonometric sine of x (in radians)
cos(x)	trigonometric cosine of x (in radians)
tan(x)	trigonometric tangent of x (in radians)
exp(x)	exponential function
log(x)	natural logarithm of x (base e)
log10(x)	logarithm of x to base 10
abs(x)	absolute value (unsigned)
ceil(x)	rounds x up to nearest integer
floor(x)	rounds x down to nearest integer
pow(x,y)	x raised to power y

Functions(cont.)

- A function **declaration** tells the compiler about a function's name, return type, and parameters.
- A function **definition** provides the actual body of the function.

- A function is knows as with various names like a method or a sub-routine or a procedure etc

Functions(cont.)

- **Defining a Function:**
  - The general form of a C++ function definition is as follows:

    return_type function_name(parameter list)

    { body of the function }

    - 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 Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
    - Parameters
    - Function Body

Functions(cont.)

- **Example:**
  - Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
int max(int num1, int num2)
{
    int result;
   if (num1 > num2)
        result = num1;
   else result = num2;
        return result;
}
```

Functions(cont.)

Function Declarations

- A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration has the following parts:
  - return_type function_name(parameter list);
- Example:
  - int max(int num1, int num2); Parameter names are not importan 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.

Functions(cont.)

- **Calling a Function:**
  - While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.
  - When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.
  - To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

Functions(cont.)

- **Calling a Function:Example**

```
#include <iostream>
using namespace std;
// function declaration
int max(int num1, int num2);
int main ()
{      int a = 100; int b = 200; int ret;
       ret = max(a, b); // calling a function to get max value.
        cout << "Max value is : " << ret << endl; return 0;
}
int max(int num1, int num2)
       {int result;
       if (num1 > num2) result = num1;
       else result = num2;
        return result;}
```

Functions(cont.)

- **Function Prototypes**
  - A function prototype eliminates the need to place a function definition before all calls to the function.
  - Function prototypes are also known as function declarations.
  - Note: You must either place the function definition or the function prototype ahead of all calls to the function. Otherwise the program will not compile.

Functions(cont.)

- **Function Prototypes**

```
#include <iostream>
using namespace std;
// Function prototypes
 void first();
 void second();
 int main()
     { cout << "I am starting in    function
     main.\n";
        first(); // Call function first
       second(); // Call function second
     cout << "Back in function main
     again.\n";
return 0;}
```

```
// Definition of function first *
 void first()
{cout << "I am now inside the
function first.\n";}
// Definition of function second *
 void second()
 {
cout << "I am now inside the
function second.\n";
}
```

Functions(cont.)

Sending Data into a Function

- Values that are sent into a function are called arguments.
- In the following statement the function pow is being called with two arguments, 2 and 4, passed to it:

```
result = pow(2, 4);
```

- A parameter is a special variable that holds a value being passed as an argument into a function.

Functions(cont.)

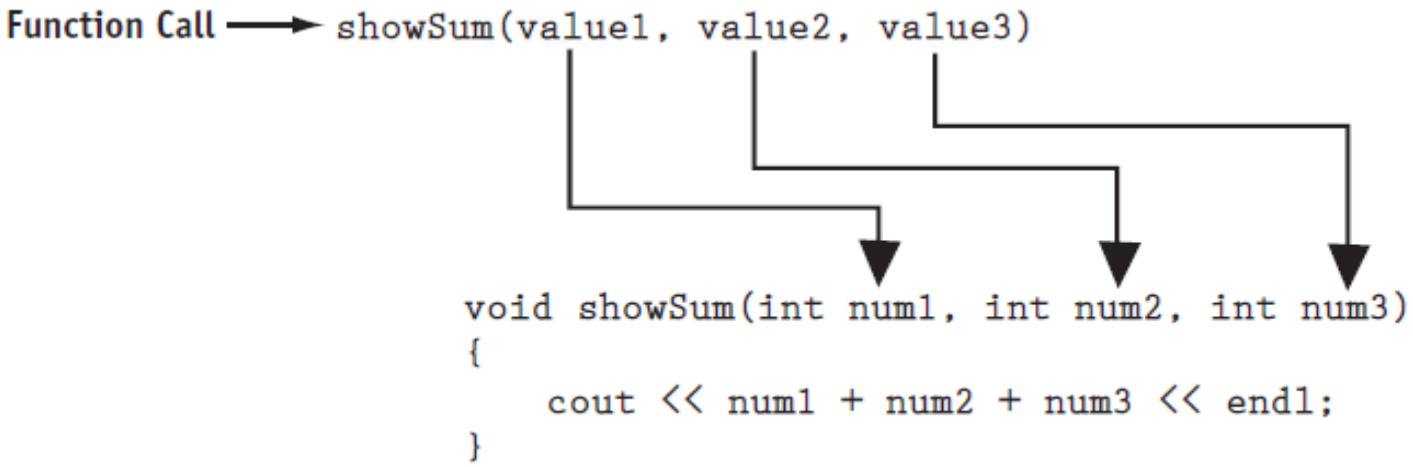
- **Sending Data into a Function :Example**

```
#include <iostream>
 using namespace std;
// Function prototype
 void displayValue(int);
int main()
    { cout << "I am passing several values to displayValue.\n";
    displayValue(5); // Call displayValue with argument 5
    displayValue(10); // Call displayValue with argument 10
    cout << "Now I am back in main.\n"; return 0; }
 void displayValue(int num)
{cout << "The value is " << num << endl;}
```

Functions(cont.)

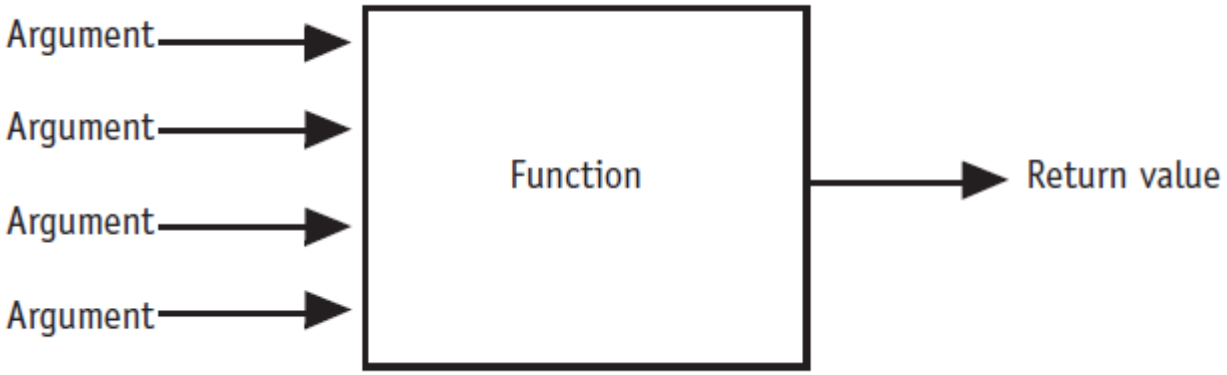
Sending Data into a Function :

- When a function with multiple parameters is called, the arguments are passed to the parameters in order



Functions(cont.)

- **Returning a value from a function**



Functions(cont.)

```
#include <iostream>
 using namespace std;
// Function prototype
int square(int);
int main()
{ int number, result;
      cout << "Enter a number and I will
     square it: ";
      cin >> number;
      result = square(number);
      cout << number << " squared is " <<
     result << endl;
 return 0;}
```

```
int square(int number)
{
return number * number;
}
```

Functions(cont.)

- **Returning a Boolean Value**
  - **Functions may return true or false values.**

```
#include <iostream>
 using namespace std;
// Function prototype
  bool isEven(int);
  int main()
{ int val;
 cout << "Enter an integer and I will tell you ";
 cout << "if it is even or odd:";
cin >> val;
if (isEven(val))
 cout << val << " is even.\n";
  else
    cout << val << " is odd.\n";
return 0; }
 bool isEven(int number)
{  if (number % 2==0)
      return true;
   else
        return false;  }
```

Functions(cont.)

- **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.
  - The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

Functions(cont.)

Function Arguments:

- While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
Call by reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Functions(cont.)

Passing Data by Value

- The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, 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

Functions(cont.)

- **Passing Data by Value Example**

```
// function definition to swap the values.
 void swap(int x, int y)
{ int temp;
      temp = x;
      x = y;
      y = temp;
}
```

```
#include <iostream>
using namespace std;
int main ()
 {int a = 100; int b = 200;
 cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl;
// calling a function to swap the values.  swap(a, b);
cout << "After swap, value of a :" << a << endl; cout << "After swap, value of b :" << b << endl;
 return 0; }
```

Functions(cont.)

- **output**

  Before swap, value of a :100

  Before swap, value of b :200

  After swap, value of a :100

  After swap, value of b :200

- **Which shows that there is no change in the values though they had been changed inside the function**

Functions(cont.)

- **call by reference**
  - The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

Functions(cont.)

- **call by reference**
  - **Example**

```
// function definition to swap the values.
 void swap(int &x, int &y)
{ int temp;
      temp = x;
      x = y;
      y = temp;
}
```

```
#include <iostream>
using namespace std;
int main ()
 {int a = 100; int b = 200;
 cout << "Before swap, value of a :" << a << endl;
cout << "Before swap, value of b :" << b << endl;
// calling a function to swap the values.  swap(a, b);
cout << "After swap, value of a :" << a << endl; cout << "After swap, value of b :" << b << endl;
 return 0; }
```

Functions(cont.)

- **output**

  Before swap, value of a :100

  Before swap, value of b :200

  After swap, value of a :200

  After swap, value of b :100

Functions(cont.)

Default Values for Parameters:

- It's possible to assign *default arguments* to function parameters. A default argument is passed to the parameter when the actual argument is left out of the function call.
- The default arguments are usually listed in the function prototype.

Here is an example:

```
void showArea(double = 20.0, double = 10.0);
```

Functions(cont.)

- **Default Values for Parameters:**
  - 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.

Functions(cont.)

- **Default Values for Parameters:Example**

```
#include <iostream>
using namespace std;
int sum(int a, int b=20)
{ int result; result = a + b; return (result); }
int main ()
{int a = 100; int b = 200; int result;
result = sum(a, b); // calling a function
cout << "Total value is :" << result << endl;
result = sum(a);
 cout << "Total value is :" << result << endl; return 0; }
```

Functions(cont.)

Passing Arrays as Function Arguments

- If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.
- Way-1
  - Formal parameters as a pointer as follows:

```
void myFunction(int *param) { . . . }
```

Functions(cont.)

- **Passing Arrays as Function Arguments**
  - Way-2
    - Formal parameters as a sized array as follows:

```
void myFunction(int param[10]) { . . . }
```

  - Way-3
    - Formal parameters as an unsized array as follows:

```
void myFunction(int param[]) { . . . }
```

Functions(cont.)

- **Passing Arrays as Function Arguments :Example**

```
double getAverage(int arr[], int size)
{
    int i, sum = 0;
    double avg;
     for (i = 0; i < size; ++i)
    { sum += arr[i]; }
    avg = double(sum) / size;
    return avg;
 }
```

Functions(cont.)

- **Passing Arrays as Function Arguments :Example**

//calling

```
#include <iostream>
using namespace std;
int main ()
{
      int balance[5] = {1000, 2, 3, 17, 50};
     double avg;
      avg = getAverage( balance, 5 )
     cout << "Average value is: " << avg << endl;
 return 0; }
```

Functions(cont.)

Return array from functions

- If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction() { . . . }
```

Thanks