



In this lecture you will learn:

- **How we can Passing Arguments to Function**
- **References**
- **Calling by Value**
- **Calling by Reference**
- **Recursion function**
- **Function overloading**
- **Functions with default parameters**

Passing Arguments to Function

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.

Example 1: (Calling by value):

```
#include <iostream>
```

```
using namespace std;
```

```
void funcValueParam(int num); // calling by value
```

```
int main()
{
    int number = 6;                                     //Line 1
    cout << "Line 2: Before calling the function "
    << "funcValueParam, number = " << number<< endl;    //Line 2
    funcValueParam(number);                             //Line 3
    cout << "Line 4: After calling the function "
    << "funcValueParam, number = " << number
    << endl;                                             //Line 4
    return 0;
}

void funcValueParam(int num)
{
    cout << "Line 5: In the function funcValueParam, "
    << "before changing, num = " << num
    << endl;                                           //Line 5
    num = 15;                                          //Line 6
    cout << "Line 7: In the function funcValueParam, "
    << "after changing, num = " << num
    << endl;                                           //Line 7
}
```

Sample Run:

Line 2: Before calling the function funcValueParam, number = 6

Line 5: In the function funcValueParam, before changing, num = 6

Line 7: In the function funcValueParam, after changing, num = 15

Line 4: After calling the function `funcValueParam`, `number = 6`

Reference Variables

A reference is an alias, a synonym for another variable. It is declared by using the reference operator **&** appended to the reference's type. Of what use is such an alias? The main use for a reference variable is as a formal argument to a function. If you use a reference as an argument, the function works with the original data instead of with a copy.

Creating a Reference Variable

C++ use the **&** symbol to indicate the address of a variable. C++ assigns an additional meaning to the **&** symbol and presses it into service for declaring References. For example, to make **rodents** an alternative name for the variable **rats**, you could do the following:

```
int rats;
```

```
int & rodents = rats; // makes rodents an alias for rats
```

Reference Arguments

One of the most important uses for references is in passing arguments to functions. Passing arguments by value is useful when the function does not need to modify the original variable in the calling program. In fact, it offers insurance that the function cannot harm the original variable. Passing arguments by reference uses a different mechanism. Instead of a value being passed to the function, a reference to the original variable, in the calling program, is passed. (It's actually the memory address of the variable that is passed, although you don't need to know this.)

- **An important advantage of passing by reference**
 - The function can access the actual variables in the calling program.
 - This provides a mechanism for passing more than one value from the function back to the calling program.

Example 2: (Calling by reference)

```
//This program reads a course score and prints the  
//associated course grade.  
  
#include <iostream>  
  
using namespace std;  
  
void getScore(int& score);  
  
void printGrade(int score);  
  
int main()  
  
{  
  
int courseScore;  
  
cout << "Line 1: Based on the course score, \n"  
<< " this program computes the "  
<< "course grade." << endl; //Line 1  
  
getScore(courseScore); //Line 2  
  
printGrade(courseScore); //Line 3
```

```
return 0;

}

void getScore(int& score)
{
    cout << "Line 4: Enter course score: ";           //Line 4
    cin >> score;                                     //Line 5
    cout << endl << "Line 6: Course score is "
    << score << endl;                                //Line 6
}

void printGrade(int cScore)
{
    cout << "Line 7: Your grade for the course is ";   //Line 7
    if (cScore >= 90)                                  //Line 8
        cout << "A." << endl;
    else if (cScore >= 80)
        cout << "B." << endl;
    else if (cScore >= 70)
        cout << "C." << endl;
    else if (cScore >= 60)
        cout << "D." << endl;
```

else

cout << "F." << endl;

}

Sample Run: In this sample run, the user input is shaded.

Line 1: Based on the course score,

this program computes the course grade.

Line 4: Enter course score: 85

Line 6: Course score is 85

Line 7: Your grade for the course is B.

Example 3: (the difference between Calling by reference and calling by value)

#include <iostream>

using namespace std;

void funOne(int a, int& b, char v);

void funTwo(int& x, int y, char& w);

int main()

{

int num1, num2;

char ch;

num1 = 10;

//Line 1

num2 = 15;

//Line 2

ch = 'A';

//Line 3

```
cout << "Line 4: Inside main: num1 = " << num1
<< ", num2 = " << num2 << ", and ch = "
<< ch << endl;                                     //Line 4
funOne(num1, num2, ch);                             //Line 5
cout << "Line 6: After funOne: num1 = " << num1
<< ", num2 = " << num2 << ", and ch = "
<< ch << endl;                                     //Line 6
funTwo(num2, 25, ch);                               //Line 7
cout << "Line 8: After funTwo: num1 = " << num1
<< ", num2 = " << num2 << ", and ch = "
<< ch << endl;                                     //Line 8
return 0;
}
void funOne(int a, int& b, char v)
{
    int one;
    one = a;                                         //Line 9
    a++;                                           //Line 10
    b = b * 2;                                     //Line 11
    v = 'B';                                       //Line 12
    cout << "Line 13: Inside funOne: a = " << a
    << ", b = " << b << ", v = " << v
    << ", and one = " << one << endl;           //Line 13
}
void funTwo(int& x, int y, char& w)
```

```
{
x++;                                     //Line 14
y = y * 2;                             //Line 15
w = 'G';                               //Line 16
cout << "Line 17: Inside funTwo: x = " << x
<< ", y = " << y << ", and w = " << w
<< endl;                             //Line 17
}
```

Sample Run:

Line 4: Inside main: num1 = 10, num2 = 15, and ch = A

Line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10

Line 6: After funOne: num1 = 10, num2 = 30, and ch = A

Line 17: Inside funTwo: x = 31, y = 50, and w = G

Line 8: After funTwo: num1 = 10, num2 = 31, and ch = G

Recursive Functions

The technique of a function calling itself is called recursion. The obvious problem is the same one for infinite loops: If a function calls itself, when does it ever stop? The problem is easily solved, however, by putting in some mechanism for stopping.

Example 4: (factorial function by using recursion technique)

```
#include<iostream>
using namespace std;
int factorial(int);
```



```
int main()
{
    int num;
    cout<<"Enter the number you want to finde the factorial to it:\n";
    cin>>num;
    cout<<" the factorial of "<<num<<" is "<<factorial(num)<<endl;
    return 0;
}

int factorial(int x)
{
    if (x <= 1)
        return 1;
    return x * factorial(x - 1);
}
```

Function Overloading

In a C++ program, several functions can have the same name. This is called function overloading, or overloading a function name. Before we state the rules to overloading a function, let us define the following:

Two functions are said to have different formal parameter lists if both functions have:

- A different number of formal parameters or

- If the number of formal parameters is the same, then the data type of the formal parameters, in the order you list them, must differ in at least one position.

Example 5: (overloading function)

```
#include <iostream>
#include <string>
using namespace std;
void larger(int x, int y);
void larger(char first, char second);
void larger(double u, double v);
void larger(string first, string second);
int main()
{
    int x, y;
    char f, s;
    double d, g;
    string st1, st2;
    larger (6, 8);
    larger ('a', 'b');
    larger (12.6, 15.8);
    larger ("compuetr", "programming");
    return 0;
}
void larger(int x, int y)
```

```
{  
    cout<<" This is the first function (int , int)\n";  
    cout<<x + y<<endl;  
}  
void larger(char first, char second)  
{  
    cout<<" This is the second function (char , char)\n";  
    cout<<first + second<<endl;  
}  
void larger(double u, double v)  
{  
    cout<<" This is the third function (double , double)\n";  
    cout<<u + v<<endl;  
}  
void larger(string first, string second)  
{  
    cout<<" This is the fourth function (string , string)\n";  
    cout<<first + second<<endl;  
}
```

Functions with default parameters

Recall that when a function is called, the number of actual and formal parameters must be the same. C++ relaxes this condition for functions with

default parameters. You specify the value of a default parameter when the function name appears for the first time, such as in the prototype.

In general, the following rules apply for functions with default parameters:

- If you do not specify the value of a default parameter, the default value is used for that parameter.
- All of the default parameters must be the far-right parameters of the function.
- Suppose a function has more than one default parameter. In a function call, if a value to a default parameter is not specified, then you must omit all of the arguments to its right.
- Default values can be constants, global variables, or function calls.
- The caller has the option of specifying a value other than the default for any default parameter.
- You cannot assign a constant value as a default value to a reference parameter.

Example 6: (overloading function)

```
#include<iostream>
using namespace std;
int power(int base, int exp = 2)
{
    int r = 1;
    for (int i = 1; i <= exp; i++)
        r*=base;
    return r;
```

```
}  
  
void main()  
{  
  
    cout <<power(4,3)<<endl;  
    cout<<power(4)<<endl;//using default argument for exp.  
}
```

Exercise:

- Write a function called zeroSmaller() that is passed two int arguments by reference and then sets the smaller of the two numbers to 0. Write a main() program to exercise this function.
- Write a program that calls a function **triple_it** that takes the address of an **int** and triples the value pointed to. Test it by passing an argument n, which is initialized to 15. Print out the value of n before and after the function is called.
- By using recursion function write program to find the GCF (greatest common factors) of two numbers.

References:

- [1] Deitel, P. & Deitel, R., "C++ How to Program", Eighth Edition, Pearson Education Inc., 2012.
- [2] Stefan B., " C++ Primer Plus ", Sixth Edition, Pearson Education Inc., 2012.