

Function and Method	Function in C++	Arguments passed by value and by reference.	Overloaded functions	inline functions
Recursivity	Declaring functions	Function templates	Passing Parameters by Value or by reference	Using Command-Line Arguments In C++

## Function and Method

- C++ was developed as an object-oriented extension of C. (In fact, an early version of C++ was called "C with Classes".) As such, it is a hybrid of two distinct programming paradigms: the procedural paradigm and the object-oriented paradigm. Java, on the other hand, was designed from the ground up as an object-oriented language.
- In Java everything in a program is part of some **class**. In C++, it is possible to have "top-level" **variables** and **functions** which are declared outside of any **class**. Further, **main()** must be top-level.
- As far as the C++ standard is concerned, there is no such thing as a "method". This terminology is used in other OO languages (e.g. Java) to refer to **member functions** of a class. In common usage, you'll find that most people will use "method" and "function" more or less interchangeably.
- In C++ a function is usually meant to mean a **free-function**, which is not the member of a class. When it is part of a class it is more commonly called **member functions** than **methods**. A member in C++ can be **private**, **protected**, **private**, **virtual**, **pure virtual**
- In Java free function cannot exist, all members must be within some class and they are called **method**.

## Function in C++

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

**type name ( parameter1, parameter2, ...) { statements }**

where:

- type is the **data type returned**, name is the identifier for calling the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, and which acts within the function as a regular **local** variable.
- statements is the function's body. It is a block of statements surrounded by braces { }.

```
#include <iostream>
using namespace std;
int addition (int a, int b)
{
    int r;    r=a+b;
    return (r);
}
```

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.

```

}
int main ()
{
    int z;    z = addition (5,3);
    cout << "The result is " << z;
    return 0;
}

```

**void** can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function printmessage could have been declared as: **void printmessage (void)**

## Arguments passed by value and by reference.

The C and C++ language are a "call by value" language, which means that the called function is given a copy of its arguments, and doesn't know their addresses.

For example, suppose we call a function **addition** using the following code:

```

int x=5, y=3, z;
z = addition ( x , y );    // we are passing the values of x and y, i.e. 5 and 3 respectively, but not the
                           // variables x and y themselves.

```

When the **addition** function will receive the parameters a and b, the local variables a and b in that function 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.

But there might be some cases where you 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:

```

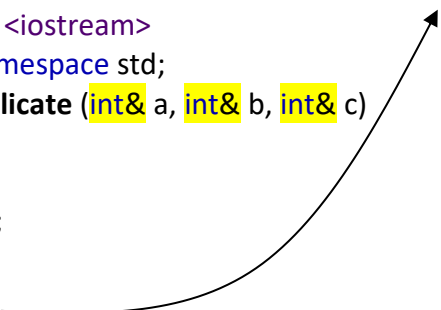
#include <iostream>
using namespace std;
void duplicate (int& a, int& b, int& c)
{
    a*=2;
    b*=2;
    c*=2;
}

```

```

int main ()
{
    int x=1, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << ", y=" << y << ", z=" << z;
    return 0;
}
x=2, y=6, z=14

```



- Here the type of each parameter was followed by an ampersand sign (&). This indicates that 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 of these parameters inside the function will have an effect in their counterpart variables passed as arguments in the call to the

## function.

- In above example shows that values stored in x, y and z after the call to duplicate are doubled.
- If it is called by value, the output on screen of our program would have been the values of x, y and z without having been modified.
- Passing by reference is also an effective way to allow a function to return more than one value.

## What's the difference between passing by reference vs. passing by value?

- Passing by reference means the called functions' parameter will be the same as the callers' passed argument (not the value, but the identity - the variable itself).
- Pass by value means the called functions' parameter will be a copy of the callers' passed argument.

**Default values in parameters :** Using the assignment operator default value of a parameter can be set. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
// default values in functions
#include <iostream>
using namespace std;
int divide (int a, int b=2)
{
    int r; r=a/b;
    return (r);
}
```

Here the default value of the 2<sup>nd</sup> parameter is

```
int main ()
{
    cout << divide (12);
    cout << endl;    cout << divide (20,4);
    return 0;
}
6
5
```

## Overloaded functions

In C++ two different functions can have the same name if their **parameter types or number** are different. For example:

```
// overloaded function
#include <iostream>
using namespace std;
int operate (int a, int b)
{
    return (a*b);
}
float operate (float a, float b)
{
    return (a/b);
}
```

```
int main ()
{
    int x=5,y=2;
    float n=5.0,m=2.0;
    cout << operate (x,y);
    cout << "\n";
    cout << operate (n,m);
    cout << "\n";
    return 0;
}
10
2.5
```

The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. Although same name the second function has a different behavior: it divides one parameter by the other.

Notice that a function **cannot be overloaded only by its return type.**

## inline functions

The **inline specifier** indicates the compiler that **inline substitution** is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the **code generated by the function body is inserted at each point the function is called, instead of being inserted only once** and perform a regular call to it, which generally involves some **additional overhead in running time**.

```
inline type name ( arguments ... ) { instructions ... }
```

and the call is just like the call to any other function. You do not have to include the **inline keyword when calling the function**, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that **inline is preferred for this function**.

## Recursivity

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the **factorial** of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$  more concretely, 5! (factorial of 5) would be:  
 $5! = 5 * 4 * 3 * 2 * 1 = 120$

and a recursive function to calculate this in C++ could be:

```
// factorial calculator
#include <iostream>
using namespace std;
long factorial (long a)
{
    if (a > 1)
        return (a * factorial (a-1));
    else
        return (1);
}
```

```
int main ()
{
    long number;
    cout << "Please type a number: ";
    cin >> number;
    cout << number << "! = " << factorial (number);
    return 0;
}
Please type a number: 9
9! = 362880
```

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime). The results given will not be valid for values **much greater than 10! or 15!**, depending on the system you compile it.

## Declaring functions

- In our earlier examples of function we define all the functions before it is called from main(). So in code first we have written the function and then the main().
- If we reverse the order i.e. first main() then function, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

But there is an **alternative way** to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a **prototype** of the function before it is used, instead of the entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is: `type name ( argument_type1, argument_type2, ...);`

- It is identical to a function definition, except that it **does not include the body of the function itself**.
- No identifiers for parameters only the type specifiers. The inclusion of a **name for each parameter** as in the function definition is **optional** in the prototype declaration. For example :

```
int protofunction (int first, int second);  
int protofunction (int, int);
```

Anyway, including a name for each variable makes the prototype more legible.

<pre>// declaring functions prototypes #include &lt;iostream&gt; using namespace std; void odd (int a); void even (int a); int main () {     int i;     do {         cout &lt;&lt; "Type a number (0 to exit): ";         cin &gt;&gt; i;    odd (i);     } while (i!=0);     return 0; }</pre>	<pre>void odd (int a) {     if ((a%2)!=0) cout &lt;&lt; "Number is odd.\n";     else even (a); } void even (int a) {     if ((a%2)==0) cout &lt;&lt; "Number is even.\n";     else odd (a); }</pre> <p>Type a number (0 to exit): 9 Number is odd. Type a number (0 to exit): 6 Number is even.</p>
---	---

Included Prototype first and declaration after main

If none of the two functions had been previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason). Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by **declaring all functions prototypes at the beginning of a program**.

## Function templates

While functions and classes are powerful and flexible tools for effective programming, in certain cases they can also be somewhat limiting because of C++'s requirement that you **specify the type of all parameters**.

For example, let's say you wanted to write a function to calculate the maximum of two numbers. You might do so like this:

<pre>int max(int nX, int nY) {     return (nX &gt; nY) ? nX : nY; }</pre>	<p>What happens later when you realize your max() function needs to work with doubles? Traditionally, the answer would be to overload the max() function and create a new version that works with doubles:</p> <pre>double max(double dX, double dY) {     return (dX &gt; dY) ? dX : dY; }</pre>
---	---

- Observe that code are same for all sorts of different types: chars, ints, doubles.
- Having to specify different “flavors” of the same function where the only thing that changes is the type of the parameters can become a **severe maintenance headache and time-waster**.

Wouldn't it be nice if we could write one version of max() that was able to work with parameters of ANY type? This is where function templates come in!



- English meaning : “**a template is a model that serves as a pattern for creating similar objects**”.
- In C++, function templates are functions that **serve as a pattern for creating other similar functions**. The basic idea behind function templates is to create a function without having **to specify the exact type(s) of some or all of the variables**. Instead, we define the function using placeholder types, called **template type parameters**.
- When you call a **template function**, the compiler replaces the **placeholder types with the actual variable types in your function call!** Using this methodology, the compiler can **create multiple “flavors” of a function from one template!**

For the **max** function above, let's create a function template. We are going to replace specific types with placeholder types. Let's call our this placeholder type “Type”. You can name your placeholder types almost anything you want, so long as it's not a reserved word. Here's our new function with a placeholder type:

```
Type max(Type tX, Type tY)
{
    return (tX > tY) ? tX : tY;
}
```

However, it won't compile because the compiler doesn't know what “**Type**” means!

In order to tell the compiler that Type is meant to be a **placeholder type**, we need to formally tell the compiler that Type is a template type parameter. This is done using what is called a template

parameter declaration:

```
template <typename Type> // this is the template parameter declaration
Type max(Type tX, Type tY)
{
    return (tX > tY) ? tX : tY;
}
```

For calling this template function  
int k=max<int>(i,j);  
double m = max<double>(p,q);

**keyword template** -- this tells the compiler that what follows is going to be a list of template parameters. We place all of our parameters inside **angled brackets (<>)**. To create a template type parameter, use either the keyword **typename** or **class**. After the typename or class keyword, all that's left is to pick a **name for your placeholder type**. Traditionally, with functions that have only one template type parameter, the name "Type" (often shortened to "T") is used. If the template function uses multiple template type parameter, they can be separated by commas:

```
template <typename T1, typename T2>
// template function here
```

**We will discuss it in more detail in Template section**

## Using Command-Line Arguments

### In C++

Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system.

Full declaration of the main function looks like :

```
int main ( int argc, char *argv[] )
```

- 1<sup>st</sup> argument is number of command line arguments
- other argument is a full list of all of the command line arguments.

The array of character pointers is the listing of all the arguments.

- argv[0] is the name of the program, or an empty string if the name is not available.
- After that, every element number less than **argc** is a **command line argument**. You can use each argv element just like a string.
- argv[argc] is a null pointer.

### Example

- argv[0] is the name of program file. So **argc** is always **at least 1**.
- You can run this program in terminal with several arguments, each argument is separated by the **space**. Then the program will print the arguments in the terminal.

```
#include<iostream.h>
#include<conio.h>
```

```
void main(int argc, char* argv[])
{
clrscr();
cout<<"\n Program name : \n"<<argv[0];
cout<<"1st arg : \n"<<argv[1];
cout<<"2nd arg : \n"<<argv[2];
cout<<"3rd arg : \n"<<argv[3];
cout<<"4th arg : \n"<<argv[4];
cout<<"5th arg : \n"<<argv[5];
getch();
}
```

Compile program

Execute program with arguments

#### Output

```
C:/TC/BIN>gcc mycmd.cpp
C:/TC/BIN>mycmd this is a program
Program name : c:/tc/bin/mycmd.cpp
1st arg : this
2nd arg : is
3rd arg : a
4th arg : program
5th arg : (null)
```