IEEE SCT SB

IEEE

Code in 10 days

Day 5

dotSlash
Workshop series

Topics for Today

- Functions
- Advantages of Functions
- Types of Functions

Functions

- A named unit of a group of program statements, which can be invoked from other parts of a program, as and when required.

- *Building blocks* of C++

- It is a type of subprogram–

  i.e. a sequence of instructions whose execution is invoked from one or more remote locations in a program, with the expectation that when its execution is complete, execution resumes at the instruction after the one that invoked the subprogram.

Advantages of Functions

- Increases code readability.
- Avoid code repetition.
- Divide a complex program into simpler ones.
- Reduce chances of errors
- Reduce program size.

Types of Functions

There are mainly two types of functions:

- Built-in Functions

These functions are part of the compiler package, they are part of the standard library.

e.g. exit( ), sqrt( ), pow( ), etc.

- User-defined Functions

These are the functions that are created by the programmer. They are created as per the requirements of the program.

Function Definition

```
type function_name (parameter list)
{
            body of the function
}
```

- *type* specifies the type of value that the return statement of the function returns.
- By default, the return type is assumed to be int.
- *parameter list* is a comma-separated list of variables of a function- referred to as its *arguments.*
- A function definition must have a return statement.

Function Definition

- The parameter list can be *open*, i.e. it can have any number of arguments.

    type function_name(...)

- The general format of the parameter declaration list for a function is:

    function_name(*type* var_name1, *type* var_name2, ..., *type* var_namen)

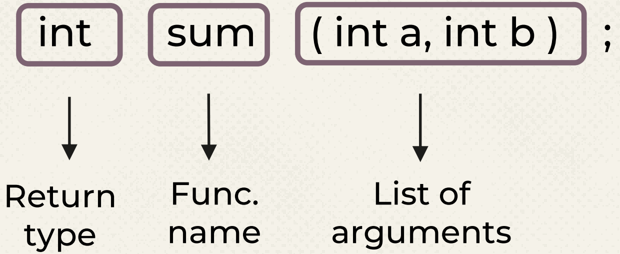
```
e.g. int absval(int a)          Function header
        {
                return(a<0 ? –a : a);          Body of the function
        }
```

Function Prototype

type function_name (*parameter list*);

- A function prototype looks similar to its definition, the only difference is the lack of a function body.
- Variable names are optional in the argument list
- A function prototype describes the function to the compiler, by giving details such as the type of return values, the number and type of arguments.

e.g.



int sum (int a, int b) ;

Return type

Func. name

List of arguments

Parameters

- Actual Parameters

The parameters present in the function call is called the actual parameter.

- Formal Parameters

The parameters present in the function header of the definition is called the formal parameter.

Accessing a Function

- A function is invoked by providing the function name, followed by parameters enclosed in parentheses.

- For example,

```
prototype: float area( float, float ) ;
function call: area( x, y ) ;
```

Program 1

```
//Program to find sum of two numbers
#include <iostream>
using namespace std;
int add(int, int);

int main()
{
    int sum;
    sum = add(100, 78);
    cout << "100 + 78 = " << sum << endl;
    return 0;
}

int add(int a, int b)
{
    return (a + b);
}
```

Program 2

```
#include<iostream>
using namespace std
int main( )
{
        float cube (float);
        float x, y;
        cout << “\nEnter a number: “;
        cin >> x;
        y = cube(x);
        cout <<“\nCube of “<<x<<“ is “<< y;
        return 0;
}

float cube( float a )
{
        float n;
        n = a*a*a ;
        return ( n );
}
```

Program 3

```
//Program to find the factorial of a number
#include<iostream>
using namespace std;
int fact(int);
int main()
{
int n, f;
cout<<"Enter the value of n : ";
cin>>n;
f=fact(n);
cout<<f;
return 0;
}
```

```
int fact(int N)
{
int f;
for(f=1; N>0; N--)
f=f*N;
return f;
}
```

Void

- The keyword *void* specifies that the function does not return any value.
- It is declared as:

  void function_name (parameter list);
- A void function cannot be used in an assignment statement.

- A function that does not require any parameter can be declared as follows:

  type function_name (void);

Arguments

Default Arguments

- These arguments can be made use of in case a matching argument is not passed in the call statement. They are specified at the time of function declaration.

```
e.g. float interest( float principal, int time, float rate=0.10);
            case 1:   si = interest(5000, 2);
            case 2:  si = interest(10000, 3, 0.15);
```

Note: Any argument cannot have a default value, unless all the arguments to its right have default values

```
e.g. float interest( float principal, int time=2, float rate );            //illegal
     float interest( float principal, int time=2, float rate=0.10 );   //legal
```

Arguments

Constant Arguments

- These are arguments whose values cannot be altered by any function.
- The keyword const is used to denote that an argument is a constant

e.g. int sum (const int a, int b);

Types of User-Defined Functions

- Void function with no arguments
- Void function with arguments
- Non-void function with no arguments
- Non-void function with arguments

Types of User-Defined Functions

Void function with no arguments

- This function does not send or receive any parameters, and it does not return any value.

Syntax:

void *function_name* ( )
{ }

Example:

```
void stars( )
{
   for ( int i=0; i<5; i++)
      cout << “*”;
   cout<<endl;
   return;
}
```

Void function with arguments

- This function receives some parameters, but it does not return any value.

Syntax:

void *function_name*(*argument_list*)
{ }

Example:

```
void avg( int a, int b)
{
   float s, av ;
   s = a+b ;
   av = s/2 ;
   cout<<“Average: “<< av<<endl;
   return;
}
```

Types of User-Defined Functions

Non-void function with no arguments

- This function takes no parameters, but it does return a value.

Syntax:

```
return_type function_name ( )
{
            return ( value );
}
```

Example:

```
char Grade( )
{
   char g;
   if ( p>45 )
      g = ‘P’ ;
   else
      g = ‘F’ ;
}
```

Non-void function with arguments

- This function takes some parameters, and it does return a value.

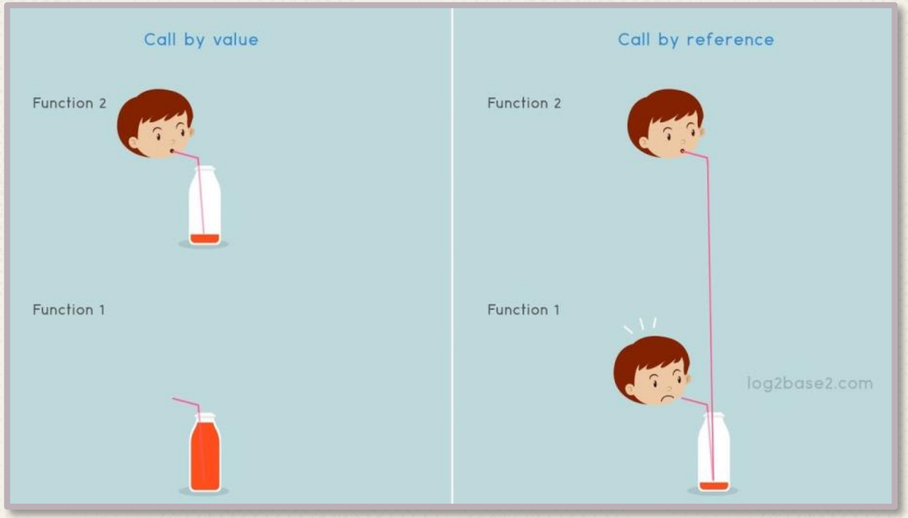
Syntax:

```
return_type function_name(argument_list)
     {
                return ( value );
     }
```

Example:

```
float avg( int a, int b)
{
   float s, av ;
   s = a+b ;
   av = s/2 ;
   return av;
}
```

Invoking a Function



A function can be invoked in two manners:

- Call by Value
- Call by Reference

Call by Value

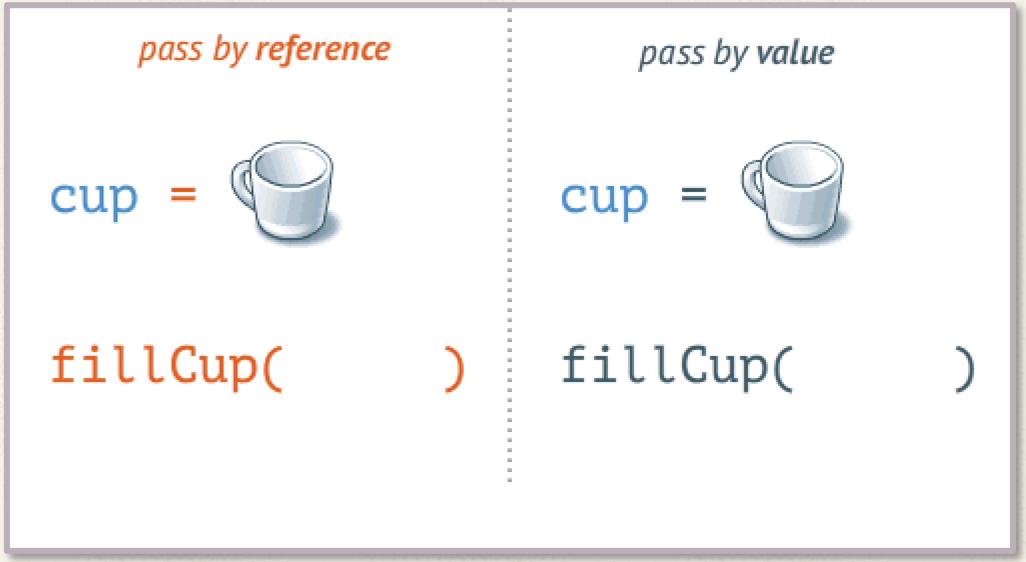
- In this method, the values of the actual parameters are copied into the formal parameters, i.e. a copy of the argument values are created, and are used.

- The main benefit of this method is that the value of the variables used to call the function cannot be altered.

Call by Reference

- In this method, a ***reference*** to the original parameters is passed to the function being called.
- Here, the changes made to the value of a variable is reflected back to the original value.

Invoking a Function



Program 4

```
//Program to illustrate call by value
 #include<iostream>
 using namespace std
 int main( )
 {
               int change(int );
               int o = 10;
               cout << “\nOriginal values:  “<< o << “\n;
               cout <<“\nValue returned from function:
               \n“<< change( o );
               cout << “\nValue after function is complete: “ << o;
               return 0;
 }
 int change ( int a )
 {
               a=20 ;
               return a ;
 }
```

Program 5

```
//Program to swap two numbers
 #include<iostream>
 using namespace std
 int main( )
 {
         void swap(int &, int &);
         int x=3, y=8;
         cout << “\nOriginal values: \n“;
         cout << “x: “<<x<<“y:”<<y<<“\n”;
         swap( x, y );
         cout <<“\nValues after swapping: \n“;
         cout << “x: “<<x<<“y:”<< y;
         return 0;
 }
 void swap( int &a, int &b )
 {
         int temp;
         temp = a;
         a=b;
         b=temp;
         cout <<“\nSwapped Values: \n“;
         cout << “x: “<<a<<“y:”<< b;
 }
```

Thank You