C++ Course Notes | Functions and pointers | Week 4

Hello everyone welcome to the weekly lecture notes

Topics to be covered:

1. Functions
2. Pointers

What are functions ?

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Why functions ?

Because using functions makes the code neat and reusable.

For example , below written code is to write a function to sum two numbers.

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

int sum(int a,int b){
    return a + b;
}

int main(){
    cout<<sum(4 , 5);
}
```

Output :

9

Explanation :

In this code, we have written a separate function named as `sum` , that accepts two parameters `a` and `b` and returns an integer, which is the sum of the passed parameters.

Components of a function :

There are mainly 3 components of any function :

- Function name
- Function return type
- Argument list (Optional)

1. Function name : This is name of the function that the user chooses. In general practice, function name does not start with uppercase character.

   For example, `myCode` , `difference` are valid function names.

2. Function return type : This defines what type of data will be returned by the function. It can be `int` , `char` , `string` , `void` etc.
3. Argument list : This is a list of parameters that we will be passing in our function. There can be any number of parameters in a function. A function can also be created without any argument list therefore this is optional.

A C++ function consist of two parts:

- **Declaration:** the return type, the name of the function, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

Note : A function must be declared before it is called. This will become more clear with the help of the below given example :

```
int main() {
  functionWithPW();
  return 0;
}

void functionWithPW() {
  cout << "Hello pwians";
}

// Error
```

The above stated code will generate an error, since `functionWithPW` is not declared before its call.

To avoid this error we can split the definition and declaration of the function in two different parts as shown in the example below:

```
//function declaration
void functionWithPW();

//main program
int main() {
  functionWithPW();
  return 0;
}

//function definition
void functionWithPW() {
  cout << "Hello pwians";
}
```

Different types of passing a parameter :

A parameter can be passed primarily in two ways :

- Pass by value
- Pass by reference

Pass 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.

For example :

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

void swap(int a,int b){
    int temp = a;
    a = b;
    b = temp;

    return;
}

int main(){
    int a = 6 , b = 8;
    cout<<"Values of a and b respectively before swap : "<<a<<" "<<b<<endl;
    swap(a,b);
    cout<<"Values of a and b respectively after swap : "<<a<<" "<<b<<endl;
}
```

Output :

```
Values of a and b respectively before swap : 6 8

Values of a and b respectively after swap : 6 8
```

Here we can see the values have not been swapped. This is because we have passed the parameters by value that means a copy of `a` and `b` have been passed and the changes made by the `swap` function are observed in those copies and not reflected back in the original variables. This is pass by value.

Pass 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.

For example,

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

void swap(int &a,int &b){
    int temp = a;
    a = b;
    b = temp;

    return;
}

int main(){
    int a = 6 , b = 8;
    cout<<"Values of a and b respectively before swap : "<<a<<" "<<b<<endl;
    swap(a,b);
    cout<<"Values of a and b respectively after swap : "<<a<<" "<<b<<endl;
}
```

Output :

```
Values of a and b respectively before swap : 6 8
```

```
Values of a and b respectively after swap : 8 6
```

Here, we have passed the address of the original variables `a` and `b`. Therefore all the changes made by the `swap` function are reflected in the original variables as well.

Formal Parameter : A variable and its type as they appear in the prototype of the function or method.

Actual Parameter : The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Formal parameters Vs Actual parameters

Actual Parameters	**Formal Parameters**
Defined when it is invoked.	Defined when the function is called.
Passed on calling a function	They will be in the called function.
Data types will not be mentioned.	The data type must be included.
They are written in the function call.	They are written in the function definition.
They may be constant values.	They can be local variables.

Scope of a variable : scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with.

Variables are of two types :

- Local variables : Variables defined within a function or block are said to be local to those functions. These variables are not accessible outside the function or code block.

  For example,

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

  void function1(){
      int a = 5 ;
  }
  void myFunction(){
      cout<<a<<" ";
  }
  int main(){
      function1();
      myFunction();
  }
  ```

  Output : `error: 'a' was not declared in this scope.`

- Global variables : they can be accessed from any part of the program. They are available through out the life time of a program. They are declared at the top of the program outside all of the functions or blocks.

  For example,

  ```
  #include<iostream>
  using namespace std;
  int b  = 10;

  void myFunction(){
     cout<<b<<endl;
  }
  int main(){
      myFunction();
      cout<<b<<endl;
  }
  ```

  Output :

  ```
  10
  10
  ```

What is a pointer ?

The pointer is a variable, it is also known as locator or indicator that points to an address of a value. The symbol of an address is represented by a pointer. In addition to creating and modifying dynamic data structures, they allow programs to emulate call-by-reference.

For example, `*pointer_name`

Dereferencing operator(*) : Dereferencing is the method where we are using a pointer to access the element whose address is being stored. We use the * operator to get the value of the variable from its address.

```
int a;

// Referencing
int *ptr=&a;

// Dereferencing
int b=*ptr;
```

Note : Arithmetic such as add, subtract can be performed on pointers as well.

Pointer Increment in C++:

```
int array[5]={2,4,6,8,11}

int *ptr = array;

ptr++; //pointer increment won't add 1 to address instead it move to the next immediate
index.
```

Pointer Decrement in C++ :

```
int array[5]={1,2,3,4,5}
int *ptr=array;
ptr++;
ptr—; //pointer decrement will decrement to one less index in the array therefore we
have first incremented to the next index so that it might not result in runtime error.
```

Similarly, pointers can be added or subtracted by any constant value as well.

Double pointers : A pointer stores the memory address of other variables. So, when we define a pointer to a pointer, the first pointer is used to store the address of the variables, and the second pointer stores the address of the first pointer. For this very reason, this is known as a **Double Pointer** or **Pointer to Pointer**.

For example,

```
#include <bits/stdc++.h>
using namespace std;

int main(){
int data = 1;

int* pointer1;

int** pointer2;

pointer1 = &data;

pointer2 = &pointer1;

cout << "Value of data :- " <<data << endl;
cout << "Value of data using single pointer :- " <<
   *pointer1 << endl;
cout << "Value of data using double pointer :- " <<
   **pointer2 << endl;
return 0;
}
```

Output :

```
Value of data :- 1

Value of data using single pointer :- 1

Value of data using double pointer :- 1
```