Practical Lecture : Function

Quick Recap

Let’s take a quick recap of previous lecture –

A)

B)

C)

D)

E)

Today’s Agenda

Today we are going to cover -

- Function
- Function Overloading and scope rules
- Inline Function
- Manipulators Functions

Let’s Get Started-

Functions

A function is block of code which is used to perform a particular task, for example let’s say you are writing a large C++ program and in that program you want to do a particular task several number of times, like displaying value from 1 to 10, in order to do that you have to write few lines of code and you need to repeat these lines every time you display values. Another way of doing this is that you write these lines inside a function and call that function every time you want to display values. This would make you code simple, readable and reusable.

Function

```
#include <iostream>
using namespace std;
                              //Function declaration
int sum(int,int);
                             //Main function
int main(){
                             //Calling the function
  cout<<sum(1,99);
  return 0;
}
                              /* Function is defined after the main method
                                                 */
int sum(int num1, int num2){
  int num3 = num1+num2;
  return num3;
}
```

Function

Function Declaration: You have seen that I have written the same program in two ways, in the first program I didn't have any function declaration and in the second program I have function declaration at the beginning of the program. The thing is that when you define the function before the main() function in your program then you don't need to do function declaration but if you are writing your function after the main() function like we did in the second program then you need to declare the function first, else you will get compilation error

Function

syntax of function declaration:

return_type function_name(parameter_list);

Function definition: Writing the full body of function is known as defining a function.

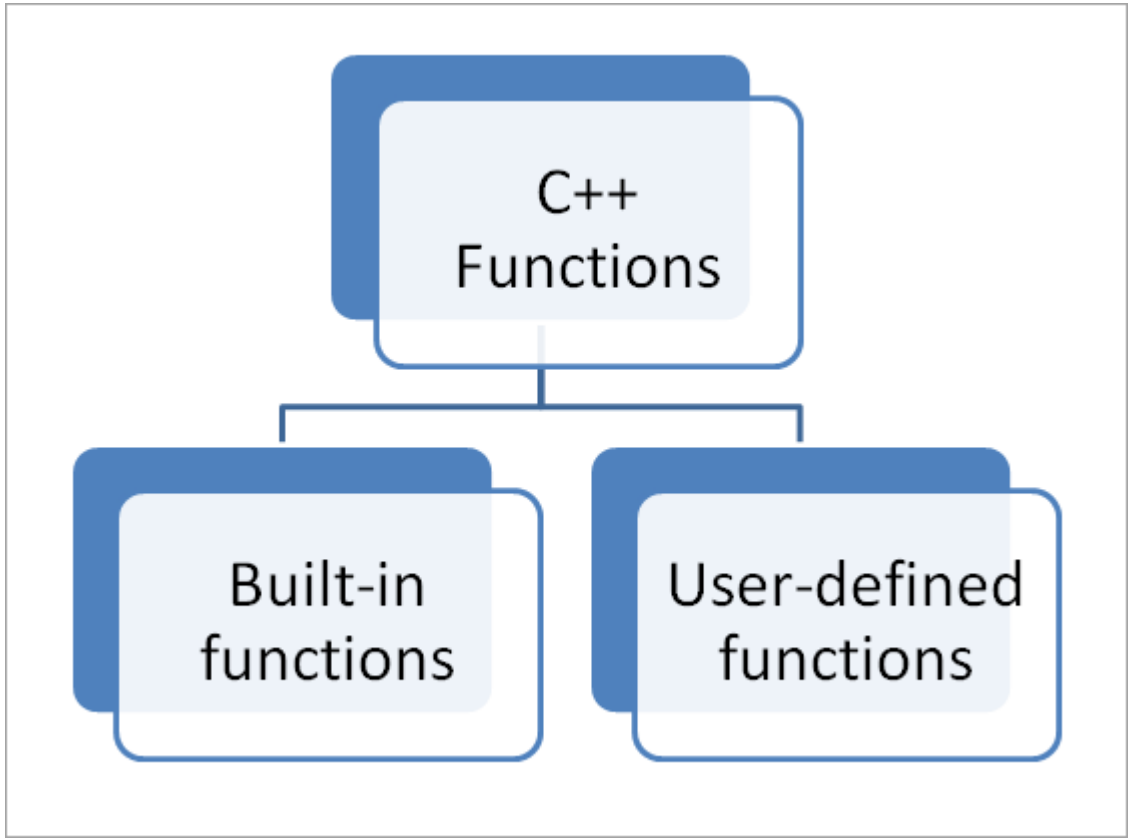
syntax of function definition:

```
return_type function_name(parameter_list) {
   //Statements inside function
}
```

Calling function: We can call the function like this:

function_name(parameters);

Function



Build In Function

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

int main() {

double num, squareRoot;
cout << "Enter number: ";
cin >> num;
squareRoot = sqrt(num);
cout << "The square root of " << num << " is: " << squareRoot;
return 0;
}
```

User Defined Function

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


void sayHello() {
cout << "Hello!";
}


int main() {


sayHello();


return 0;
```

Default Values for the parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. 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.

Consider the following example −

Default Values for the parameters

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

int sum(int a, int b = 20) {
   int result;
   result = a + b;

   return (result);
}
```

Default Values for the parameters

```
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;
   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // calling a function again as follows.
   result = sum(a);
   cout << "Total value is :" << result << endl;
   return 0;
}
```

Default Values for the parameters

Total value is :300

Total value is :120

Practice Questions In Class

- Create a calculator that takes a number, a basic math operator (+,-,*,/,^), and a second number all from user input, and have it print the result of the mathematical operation. The mathematical operations should be wrapped inside of functions.

Assignment

- A person is elligible to vote if his/her age is greater than or equal to 18. Define a function to find out if he/she is elligible to vote.

- Write a program which will ask the user to enter his/her marks (out of 100). Define a function that will display grades according to the marks entered as below:

Marks	Grade
91-100	AA
81-90	AB
71-80	BB
61-70	BC
51-60	CD
41-50	DD
<=40	Fail

Function Overloading

Function overloading is a C++ programming feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters,

Function Overloading Example 1

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

class Addition {
public:
    int sum(int num1,int num2) {
        return num1+num2;
    }
    int sum(int num1,int num2, int num3) {
       return num1+num2+num3;
    }
};
```

Function Overloading Example 1

```
int main(void) {
    Addition obj;
    cout<<obj.sum(20, 15)<<endl;
    cout<<obj.sum(81, 100, 10);
   return 0;
}
```

Ouput

```
35
191
```

Function Overloading Example 2

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

class DemoClass {
public:
    int demoFunction(int i) {
        return i;
    }
    double demoFunction(double d) {
        return d;
    }
};
```

Function Overloading Example 2

```
int main(void) {
    DemoClass obj;
    cout<<obj.demoFunction(100)<<endl;
    cout<<obj.demoFunction(5005.516);
   return 0;
}
```

Output:-

```
100
5006.52
```

Advantage

The main advantage of function overloading is to the improve the **code readability** and allows **code reusability**. In the example 1, we have seen how we were able to have more than one function for the same task(addition) with different parameters, this allowed us to add two integer numbers as well as three integer numbers, if we wanted we could have some more functions with same name and four or five arguments.

Imagine if we didn't have function overloading, we either have the limitation to add only two integers or we had to write different name functions for the same task addition, this would reduce the code readability and reusability

Inline Functions

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

Inline Functions

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

inline int Max(int x, int y) {
   return (x > y)? x : y;
}

// Main function for the program

int main() {
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;

   return 0;
}
```

Output:-

Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010

Advantage of Inline Function

The **inline functions** are a C++ enhancement feature to increase the execution time of a program. **Functions** can be instructed to compiler to make them **inline** so that compiler can replace those **function** definition wherever those are being called.

Manipulators

Manipulators are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

For example, if we want to print the hexadecimal value of 100 then we

cout<<setbase(16)<<100

Types of Manipulators

Manipulators without arguments: The most important manipulators defined by the **IOStream library** are provided below.

- **endl**: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.

- **ws**: It is defined in istream and is used to ignore the whitespaces in the string sequence.

- **ends**: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.

- **flush**: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time.

Types of Manipulators

```
#include <iostream>
#include <istream>
#include <sstream>
#include <string>

using namespace std;

int main()
{
   istringstream str("        Programmer");
   string line;
   // Ignore all the whitespace in string
   // str before the first word.
   getline(str >> std::ws, line);
```

Types of Manipulators

```
// you can also write str>>ws
   // After printing the output it will automatically
   // write a new line in the output stream.
   cout << line << endl;

   // without flush, the output will be the same.
   cout << "only a test" << flush;

   // Use of ends Manipulator
   cout << "\na";

   // NULL character will be added in the Output
   cout << "b" << ends;
   cout << "c" << endl;
   return 0;
}
```

Output

Programmer
only a test
abc

Manipulators

Manipulators with Arguments: Some of the manipulators are used with the argument like setw (20), setfill (‘*’), and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program.
For Example, you can use following manipulators to set minimum width and fill the empty space with any character you want: std::cout << std::setw (6) << std::setfill (’*’);

Manipulators

Some important manipulators in *<iomanip>* are:

- **setw (val):** It is used to set the field width in output operations.
- **setfill (c):** It is used to fill the character ‘c’ on output stream.
- **setprecision (val):** It sets val as the new value for the precision of floating-point values.
- **setbase(val):** It is used to set the numeric base value for numeric values.
- **setiosflags(flag):** It is used to set the format flags specified by parameter mask.
- **resetiosflags(m):** It is used to reset the format flags specified by parameter mask.

Manipulators

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

int main()
{
    double A = 100;
    double B = 2001.5251;
    double C = 201455.2646;

    // We can use setbase(16) here instead of hex

    // formatting
    cout << hex << left << showbase << nouppercase;
```

Manipulators

```
// actual printed part
  cout << (long long)A << endl;

  // We can use dec here instead of setbase(10)

  // formatting
  cout << setbase(10) << right << setw(15)
     << setfill('_') << showpos
     << fixed << setprecision(2);

  // actual printed part
  cout << B << endl;
```

Manipulators

```
 // formatting
    cout << scientific << uppercase
         << noshowpos << setprecision(9);

    // actual printed part
    cout << C << endl;
}
```

Output:-

```
0x64
_______+2001.53
2.014552646E+05
```

QNA Time

Any Questions ??

Any Questions??

Thank You!

See you guys in next class.