Inline Function

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.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers −

```
#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
```

Week 4: Lecture 8

C++ Function Arguments

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function-

i) Call by value

ii) Call by reference

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Call By Value

In call by value method, the called function creates its own copies of original values sent to it. Any changes, that are made, occur on the function's copy of values and are not reflected back to the calling function.

C++ allows calling a function without specifying all its arguments. In such cases, the function assigns a default value to a parameter which does not have matching arguments in the function call. Default values are specified when the function is declared. The complier knows from the prototype how many arguments a function uses for calling.

Example :

float result(int marks1, int marks2, int marks3=75);

a subsequent function call

average = result(60,70);

passes the value 60 to marks1, 70 to marks2 and lets the function use default value of 75 for marks3.

The function call

average = result(60,70,80);
passes the value 80 to marks3.

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 –

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

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

   return (result);
}
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;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total value is :300
Total value is :120
```

Call By Reference

In call be reference method, the called function accesses and works with the original values using their references. Any changes, that occur, take place on the original values are reflected back to the calling code.

Consider the following program which will swap the value of two variables.

Using call by reference using call by value

#include<iostream>
using namespace std;
void swap(int &, int &);
int main()
{
int a=10,b=20;
swap(a,b);
cout<<a<<" "<<b;
return 0;
}
void swap(int &c, int &d)
{
int t;
t=c;
c=d;
d=t;
}
output:
20 10
	#include<iostream>
	using namespace std;
	void swap(int , int);
	int main()
	{
	int a=10,b=20;
	swap(a,b);
	cout<<a<<" "<< b;
	return 0;
	}
	void swap(int c, int d)
	{
	int t;
	t=c;
	c=d;
	d=t;
	}
	output:
	10 20

Function With Examples:

Function with no return and no argument.

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

void sum();
void sum()
{
   int a=10,b=20,c;
   c=a+b;
   cout<<c;
}
int main()
{
   sum();
   return 0;
}
```

Output: 30

Function with return and no argument.

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

int sum();
int sum()
{
   int a=10,b=20,c;
   c=a+b;
return c;
}
int main()
{
   cout<<sum();
   return 0;
}
```

Output: 30

Function with no return and argument.

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

void sum(int a, int b);
void sum(int a, int b)
{
   int c;
   c=a+b;
   cout<<c;
}
int main()
{
   sum(10,20);
   return 0;
}
```

Output: 30

Function with return and argument.

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

int sum(int a, int b);
int sum(int a, int b)
{
   int c;
   c=a+b;
   return c;
}
int main()
{
   cout<<sum(10,20);
   return 0;
}
```

Output: 30

To get more information follow the url:

https://www.tutorialspoint.com/cplusplus/cpp_functions.htm

Lecture: Function Overloading and float in C++

Although polymorphism is a widely useful phenomenon in C++ yet it can be quite complicated at times. For instance, consider the following code snippet:

```
#include<iostream>
using namespace std;
void test(float s,float t)
{
    cout << "Function with float called ";
}
void test(int s, int t)
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5, 5.6);
    return 0;
}
```

It may appear that the call to the function test in main() will result in output "Function with float called" but the code gives following error:

```
In function 'int main()':
13:13: error: call of overloaded 'test(double, double)' is ambiguous
 test(3.5,5.6);
```

It's a well known fact in Function Overloading, that the compiler decides which function needs to be invoked among the overloaded functions. If the compiler can not choose a function amongst two or more overloaded functions, the situation is -" **Ambiguity** in Function Overloading".

The reason behind the ambiguity in above code is that the floating literals 3.5 **and** 5.6 **are actually treated as double by the compiler. Since compiler could not find a function with double argument and got confused if the value should be converted from double to int or float.**

Rectifying the error: We can simply tell the compiler that the literal is a float and NOT double by providing **suffix f**.

```
#include<iostream>
using namespace std;
void test(float s,float t)
{
    cout << "Function with float called ";
}
void test(int s,int t)
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5f, 5.6f); // Added suffix "f" to both values to
                      // tell compiler, it's a float value
    return 0;
}
```

Another Example:

```
using namespace std;

// Overloaded Function with
// float type parameter
void test(float f)
{
    cout << "Overloaded Function with float "
         << "parameter being called";
}

// Overloaded Function with
// double type parameter
void test(double d)
{
    cout << "Overloaded Function with double "
         << "parameter being called";
}

// Driver code
int main()
{
    // Overloaded Function called
    // with char type value
    test('a');

    return 0;
}
```

Why ambiguity occurs:

When there is no exact type match, the compiler looks for the closest match. The closest match for "test('a');" will be "void test(int a)", since it is not present, void test(double d) and void (float f)will cause ambiguity. Both are valid conversions. This confusion causes an error message to be displayed and prevents the program from compiling.

How to resolve ambiguity:
There are two ways to resolve this ambiguity:

1. *Typecast char to float.*
2. *Remove either one of the ambiguity generating functions float or double and add overloaded function with an int type parameter.*

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

// Overloaded Function with
// float type parameter
void test(float f)
{
    cout << "Overloaded Function with float "
         << "parameter being called";
}

// Overloaded Function with
// double type parameter
void test(double d)
{
    cout << "Overloaded Function with double "
         << "parameter being called";
}

// Driver code
int main()
{
    // Overloaded Function called
    // with char type value
    // typecasted to float
    test((float)('a'));

    return 0;
}
```

Practice Problem

1. **C++ Program to Make a Simple Calculator to Add, Subtract, Multiply or Divide.**

2. **Create a class named 'Student' with a string variable 'name' and an integer variable 'roll_no'. Assign the value of roll_no as '2' and that of name as "John" by creating an object of the class Student.**

3. **Write a program to print the area and perimeter of a triangle having sides of 3, 4 and 5 units by creating a class named 'Triangle' with a function to print the area and perimeter.**

4. **Write a program to print the area of two rectangles having sides (4,5) and (5,8) respectively by creating a class named 'Rectangle' with a function named 'Area' which returns the area. Length and breadth are passed as parameters to its constructor.**

Solved 01:

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

int main() {

  char op;
  float num1, num2;

  cout << "Enter operator: +, -, *, /: ";
  cin >> op;

  cout << "Enter two operands: ";
  cin >> num1 >> num2;
```

```
switch(op) {

  case '+':
    cout << num1 << " + " << num2 << " = " << num1 + num2;
    break;

  case '-':
    cout << num1 << " - " << num2 << " = " << num1 - num2;
    break;

  case '*':
    cout << num1 << " * " << num2 << " = " << num1 * num2;
    break;

  case '/':
    cout << num1 << " / " << num2 << " = " << num1 / num2;
    break;

  default:
    // If the operator is other than +, -, * or /, error message is shown
    cout << "Error! operator is not correct";
    break;
}
```

Output

```
Enter operator: +, -, *, /: -
Enter two operands: 3.4 8.4
3.4 - 8.4 = -5

  return 0;
}
```