IOOP-Unit3

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

C++ Classes/Objects

- C++ is an object-oriented programming language.
- Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.
- Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".
- A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the **class** keyword:

Example

Create a class called "MyClass":

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
```

Create a Class

- The **class** keyword is used to create a class called **MyClass**.
- The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.
- Inside the class, there is an **integer** variable **myNum** and a **string** variable **myString**. When variables are declared within a class, they are called **attributes**.
- **At last, end the class definition with a semicolon ;.**

Create an Object

- In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.
- To create an object of MyClass, specify the class name, followed by the object name.
- To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

Create an Object

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

class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

Functions in C++

Functions in C++

- 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.

Functions in C++

Advantages of Functions in C++

- It makes the program clear and easy to understand.
- Single functions can be tested easily for errors.
- It saves time from typing the same functions again and again.
- It helps to modify the program easily without changing the structure of a program.

Create a Function

- C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.
- To create (often referred to as declare) a function, specify the name of the function, followed by parentheses ():

Create a Function

Syntax

```
void myFunction() {
  // code to be executed
}
```

- myFunction() is the name of the function
- void means that the function does not have a return value. You will learn more about return values later in the next chapter
- inside the function (the body), add code that defines what the function should do

Call a Function

- Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called.
- To call a function, write the function's name followed by two parentheses () and a semicolon ;
- In the following example, myFunction() is used to print a text (the action), when it is called:

Example : Function

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

void myFunction() {
  cout << "Hello World!\n";
}

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

Classification of Functions

- Function with no argument and no return value
- Function with no argument but return value
- Function with argument but no return value
- Function with argument and return value

Function with no argument and no return value

```
# include <iostream>
using namespace std;
void prime();
int main()
{
    // No argument is passed to prime()
    prime();
    return 0;
}
// Return type of function is void because value is not returned.
void prime()
{
    int num, i, flag = 0;

    cout << "Enter a positive integer enter to check: ";
    cin >> num;
    for(i = 2; i <= num/2; ++i)
    {
        if(num % i == 0)
        {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
    {
        cout << num << " is not a prime number.";
    }
    else
    {
        cout << num << " is a prime number.";
    }
```

No arguments passed but a return value

```
#include <iostream>
using namespace std;
int prime();
int main()
{
    int num, i, flag = 0;
    // No argument is passed to prime()
    num = prime();
    for (i = 2; i <= num/2; ++i)
    {
        if (num%i == 0)
        {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
    {
        cout<<num<<" is not a prime number.";
    }
    else
    {
        cout<<num<<" is a prime number.";
    }
    return 0;
}
// Return type of function is int
int prime()
{
    int n;
    cout<<"Enter a positive integer to check: ";
    cin >> n;
    return n;
}
```

Arguments passed but no return value

```
#include <iostream>
using namespace std;
void prime(int n);
int main()
{
    int num;
    cout << "Enter a positive integer to check: ";
    cin >> num;
    // Argument num is passed to the function prime()
    prime(num);
    return 0;
}
// There is no return value to calling function. Hence, return type of
function is void. */
void prime(int n)
{
    int i, flag = 0;
    for (i = 2; i <= n/2; ++i)
    {
        if (n%i == 0)
        {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
    {
        cout << n << " is not a prime number.";
    }
    else {
        cout << n << " is a prime number.";
    }
}
```

Arguments passed and a return value.

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

int prime(int n);

int main()
{
    int num, flag = 0;
    cout << "Enter positive integer to check: ";
    cin >> num;

    // Argument num is passed to check() function
    flag = prime(num);

    if(flag == 1)
        cout << num << " is not a prime number.";
    else
        cout<< num << " is a prime number.";
    return 0;
}

/* This function returns integer value.  */
int prime(int n)
{
    int i;
    for(i = 2; i <= n/2; ++i)
    {
        if(n % i == 0)
            return 1;
    }

    return 0;
}
```

Which method is better?

- All four programs above gives the same output and all are technically correct program.
- There is no hard and fast rule on which method should be chosen.
- The particular method is chosen depending upon the situation and how you want to solve a problem.

C++ Recursion

- A function that calls itself is known as a recursive function. And, this technique is known as recursion.

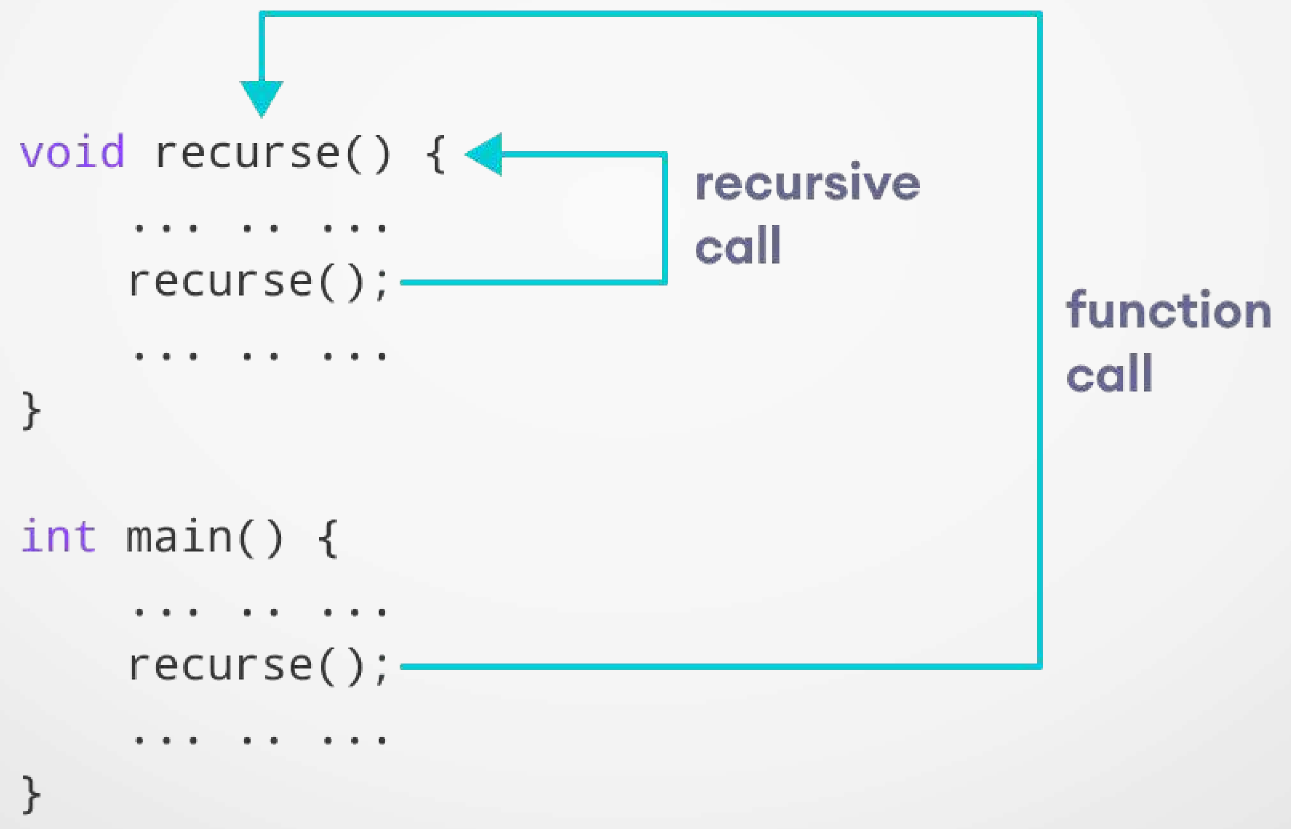
Working of Recursion in C++

-

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

C++ Recursion

- The figure below shows how recursion works by calling itself over and over again.



```
void recurse() {
    ... .. ...
    recurse();
    ... .. ...
}

int main() {
    ... .. ...
    recurse();
    ... .. ...
}
```

C++ Recursion

- The recursion continues until some condition is met.
- To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and the other doesn't.

Factorial of a Number Using Recursion

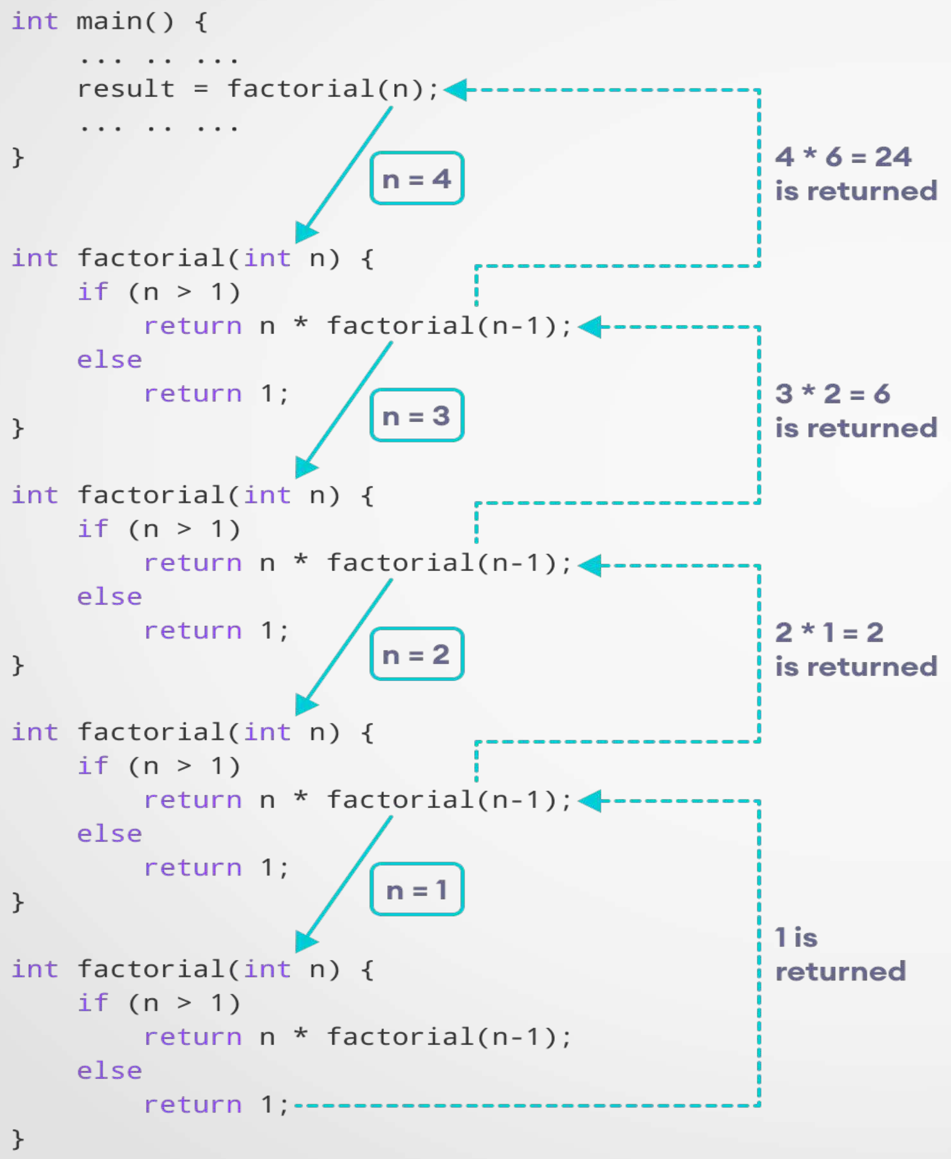
```
#include <iostream>
using namespace std;

int factorial(int);

int main() {
    int n, result;

    cout << "Enter a non-negative number: ";
    cin >> n;

    result = factorial(n);
    cout << "Factorial of " << n << " = " << result;
    return 0;
}

int factorial(int n) {
    if (n > 1) {
        return n * factorial(n - 1);
    } else {
        return 1;
    }
}
```

C++ Recursion

Output

- Enter a non-negative number: 4
- Factorial of 4 = 24

C++ Recursion



```
int main() {
    ... .. ...
    result = factorial(n);
    ... .. ...
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}

int factorial(int n) {
    if (n > 1)
        return n * factorial(n-1);
    else
        return 1;
}
```

Advantages of C++ Recursion

- It makes our code shorter and cleaner.
- Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and Tree Traversal.

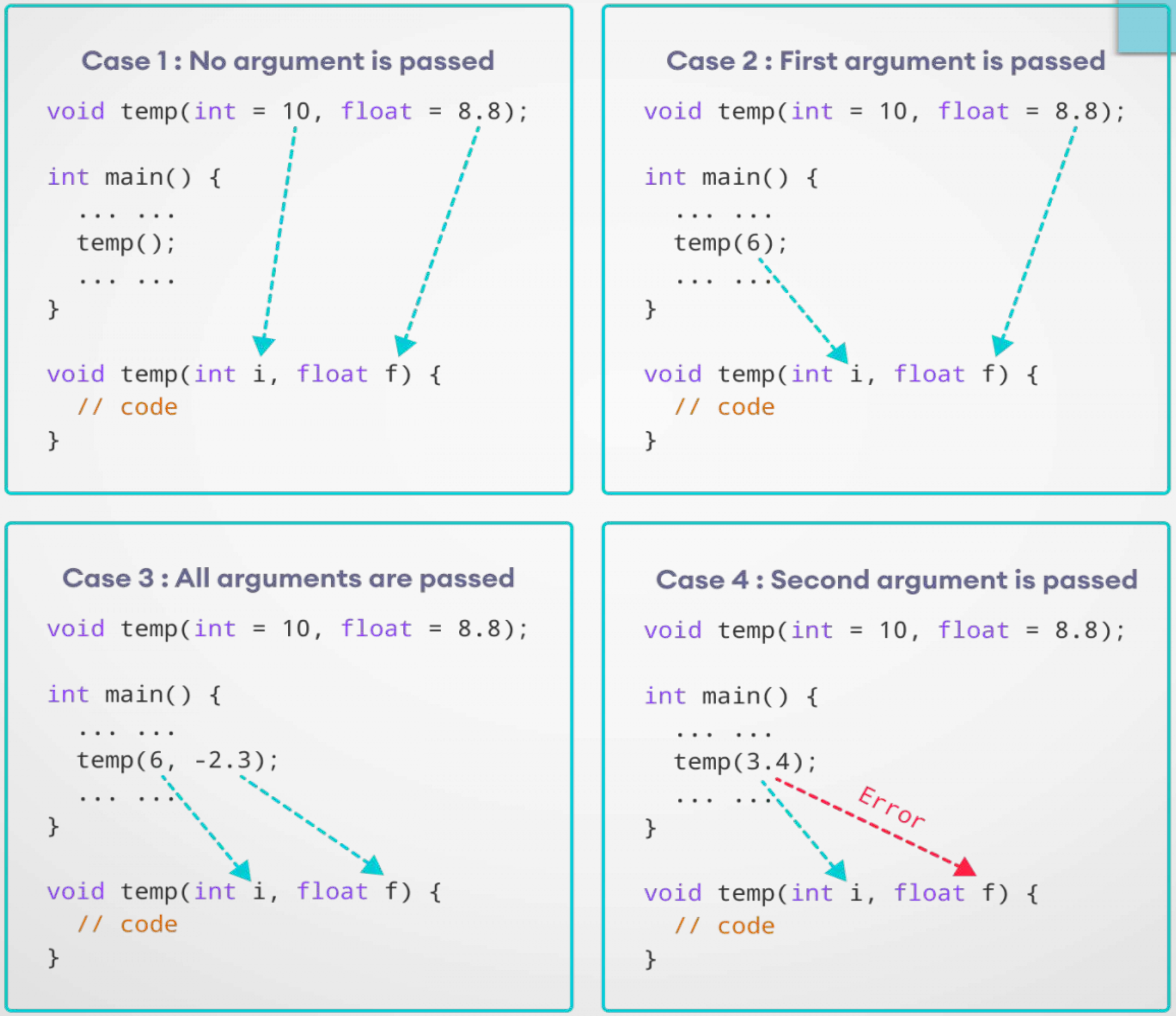
Disadvantages of C++ Recursion

- It takes a lot of stack space compared to an iterative program.
- It uses more processor time.
- It can be more difficult to debug compared to an equivalent iterative program.

Functions with Default Arguments

- In C++ programming, we can provide default values for function parameters.
- If a function with default arguments is called without passing arguments, then the default parameters are used.
- However, if arguments are passed while calling the function, the default arguments are ignored.

Working of default arguments



Case 1 : No argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp();
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 2 : First argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 3 : All arguments are passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(6, -2.3);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Case 4 : Second argument is passed

```
void temp(int = 10, float = 8.8);

int main() {
  ... ...
  temp(3.4);
  ... ...
}

void temp(int i, float f) {
  // code
}
```

Functions with Default Arguments

- **CASE 1 :** When temp() is called, both the default parameters are used by the function.
- **CASE 2 :** When temp(6) is called, the first argument becomes 6 while the default value is used for the second parameter.
- **CASE 3 :** When temp(6, -2.3) is called, both the default parameters are overridden, resulting in i = 6 and f = -2.3.
- **CASE 4 :** When temp(3.4) is passed, the function behaves in an undesired way because the second argument cannot be passed without passing the first argument.

Functions with Default Arguments

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

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
        return (x + y + z + w);
}

/* Driver program to test above function*/
int main()
{
        cout << sum(10, 15) << endl;
        cout << sum(10, 15, 25) << endl;
        cout << sum(10, 15, 25, 30) << endl;
        return 0;
}
```

Output:

```
25
50
80
```

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.
- 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.

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;
}
```

Inline Functions

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

```
Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010
```

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
- For example the parameters list of a function
- myfuncn(int a, float b) is (int, float) which is different from the function
- myfuncn(float a, int b) parameter list (float, int).
- **Function overloading is a compile-time polymorphism.**

Function overloading

- **Now that we know what is parameter list lets see the rules of overloading: we can have following functions in the same scope.**

sum(int num1, int num2)

sum(int num1, int num2, int num3)

sum(int num1, double num2)

- **The easiest way to remember this rule is that the parameters should qualify any one or more of the following conditions, they should have different type, number or sequence of parameters.**

Function overloading

- **For example:**

These two functions have different parameter type:

- **sum(int num1, int num2)**
- **sum(double num1, double num2)**

Function overloading

- **For example:**

These two have different number of parameters:

- **sum(int num1, int num2)**
- **sum(int num1, int num2, int num3)**

Function overloading

- **For example:**

These two have different sequence of parameters:

- **sum(int num1, double num2)**
- **sum(double num1, int num2)**

Note :

All of the above three cases are valid case of overloading.

We can have any number of functions, just remember that the parameter list should be different.

Function overloading

- **For example:**
- **int sum(int, int)**
- **double sum(int, int)**
- **This is not allowed as the parameter list is same.**
- **Even though they have different return types, its not valid.**

Function overloading Example

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

Output:

```
35
191
```

Function overloading Example

```
#include <iostream>
using namespace std;
class DemoClass {
public:
    int demoFunction(int i) {
        return i;
    }
    double demoFunction(double d) {
        return d;
    }
};
int main(void) {
    DemoClass obj;
    cout<<obj.demoFunction(100)<<endl;
    cout<<obj.demoFunction(5005.516);
   return 0;
}
```

Output:

```
100
5006.52
```

Advantages of Function overloading

- 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.

Scope Resolution Operator

- The scope resolution operator (::) is used for several reasons.
- For example: If the global variable name is same as local variable name, the scope resolution operator will be used to call the global variable.
- It is also used to define a function outside the class and used to access the static variables of class.

Scope Resolution Operator

1) To access a global variable when there is a local variable with same name:

```
// C++ program to show that we can access a global variable
// using scope resolution operator :: when there is a local
// variable with same name

#include<iostream>
using namespace std;

int x; // Global x

int main()
{
int x = 10; // Local x
cout << "Value of global x is " << ::x;
cout << "\nValue of local x is " << x;
return 0;
}
```

Scope Resolution Operator

Output:

```
Value of global x is 0
Value of local x is 10
```

Scope Resolution Operator

2) To define a function outside a class.

```
// C++ program to show that scope resolution operator :: is used
// to define a function outside a class
#include<iostream>
using namespace std;

class A
{
public:

// Only declaration
void fun();
};

// Definition outside class using ::
void A::fun()
{
cout << "fun() called";
}

int main()
{
A a;
a.fun();
return 0;
}
```

Scope Resolution Operator

Output:

```
fun() called
```

Scope Resolution Operator

3) To access a class’s static variables.

```
// C++ program to show that :: can be used to access static members when there is
a //local variable with same name
#include<iostream>
using namespace std;
class Test
{
	static int x;
public:
	static int y;
	// Local parameter 'a' hides class member
	// 'a', but we can access it using ::
	void func(int x)
	{
	// We can access class's static variable
	// even if there is a local variable
	cout << "Value of static x is " << Test::x;
	cout << "\nValue of local x is " << x;
	}
};
// In C++, static members must be explicitly defined like this
int Test::x = 1;
int Test::y = 2;
int main()
{
	Test obj;
	int x = 3 ;
	obj.func(x);

	cout << "\nTest::y = " << Test::y;

	return 0;
}
```

Scope Resolution Operator

Output:

```
Value of static x is 1
Value of local x is 3
Test::y = 2;
```