Introduction to C++ Functions

Md. Kamrul Islam

Lecturer, Dept. of CSE, RMU

Functions

The Top-down design approach is based on dividing the main problem into smaller tasks which may be divided into simpler tasks.

Then implementing each simple task by a subprogram or a function.

A function is a subprogram that acts on data and often returns a value.

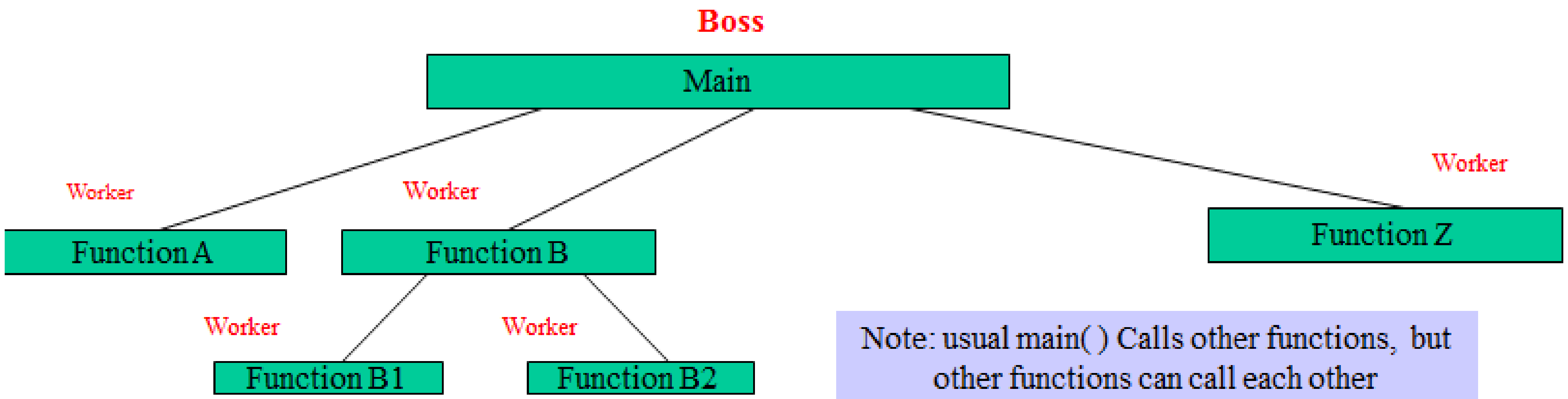
Functions are having modular approach.

Functions are the user defined data types.

About Functions in C++

- Functions invoked by a function–call-statement which consist of it’s name and information it needs (*arguments)*
- Boss To Worker Analogy

  → A Boss (the calling/caller function) asks a worker (the called function) to perform a task and return result when it is done.



Note: usual main( ) Calls other functions, but other functions can call each other

Function Types

TYPES

Functions are of two types :

- Library functions
- User defined functions

Library Functions

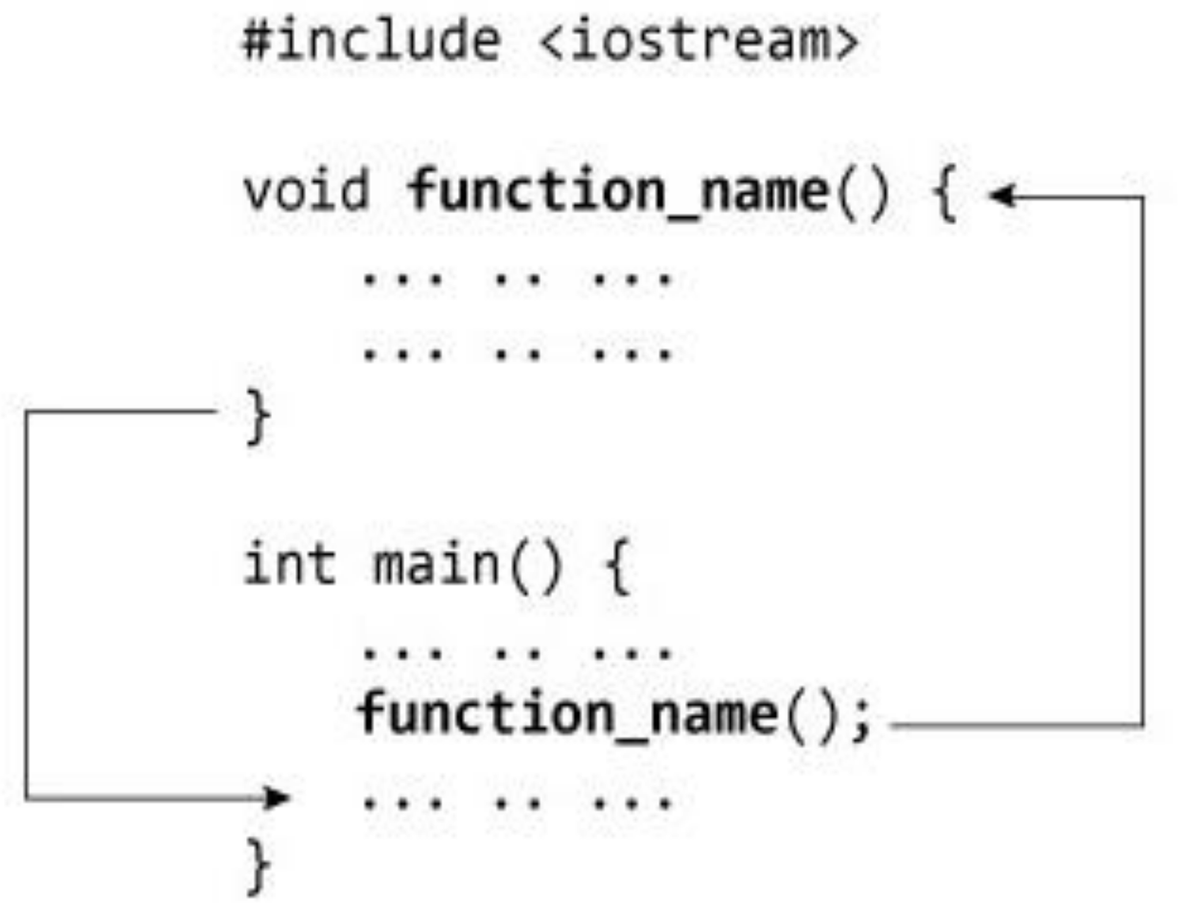
Library functions are the built-in function in C++ programming. Programmer can use library function by invoking function directly; they don't need to write it themselves.

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

int main() {
    double number, squareRoot;
    cout<<"Enter a number: ";
    cin>>number;
    squareRoot = sqrt(number);              /* sqrt() is a library function to calculate square root */
    cout<<"Square root of "<<number<<" = "<<squareRoot;
    return 0;
}
```

User Defined Functions

- ❑ C++ allows programmer to define their own function.
- ❑ A user-defined function groups code to perform a specific task and that group of code is given a name(identifier).
- ❑ When that function is invoked from any part of program, it all executes the codes defined in the body of function.

How User-defined Functions Work



```
#include <iostream>

void function_name() {
    ... .. ...
    ... .. ...
}

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

Example of User-defined Function

```
# include <iostream>
using namespace std;
int add(int, int);            //Function prototype(declaration)

int main() {
    int num1, num2, sum;
    cout<<"Enters two numbers to add: ";
    cin>>num1>>num2;
    sum = add(num1,num2);        //Function call
    cout<<"Sum = "<<sum;
    return 0;
}
int add(int a,int b) {           //Function defination
    int add;
    add = a+b;
    return add;                  //Return statement
}
```

Output

```
Output:
Enters two integers: 8
 -4
Sum = 4
```

Function Prototype(Declaration)

- In C++, function prototype is a declaration of function without function body to give compiler information about user-defined function. Function prototype in above example:

```
int add(int, int);
```

There is no body of function in prototype. Also there are only return type of arguments but no arguments.

Function Call

To execute the codes of function body, the user-defined function needs to be invoked(called).

In the above program, add(num1,num2); inside main() function calls the user-defined function. In the above program, user-defined function returns an integer which is stored in variable *add*.

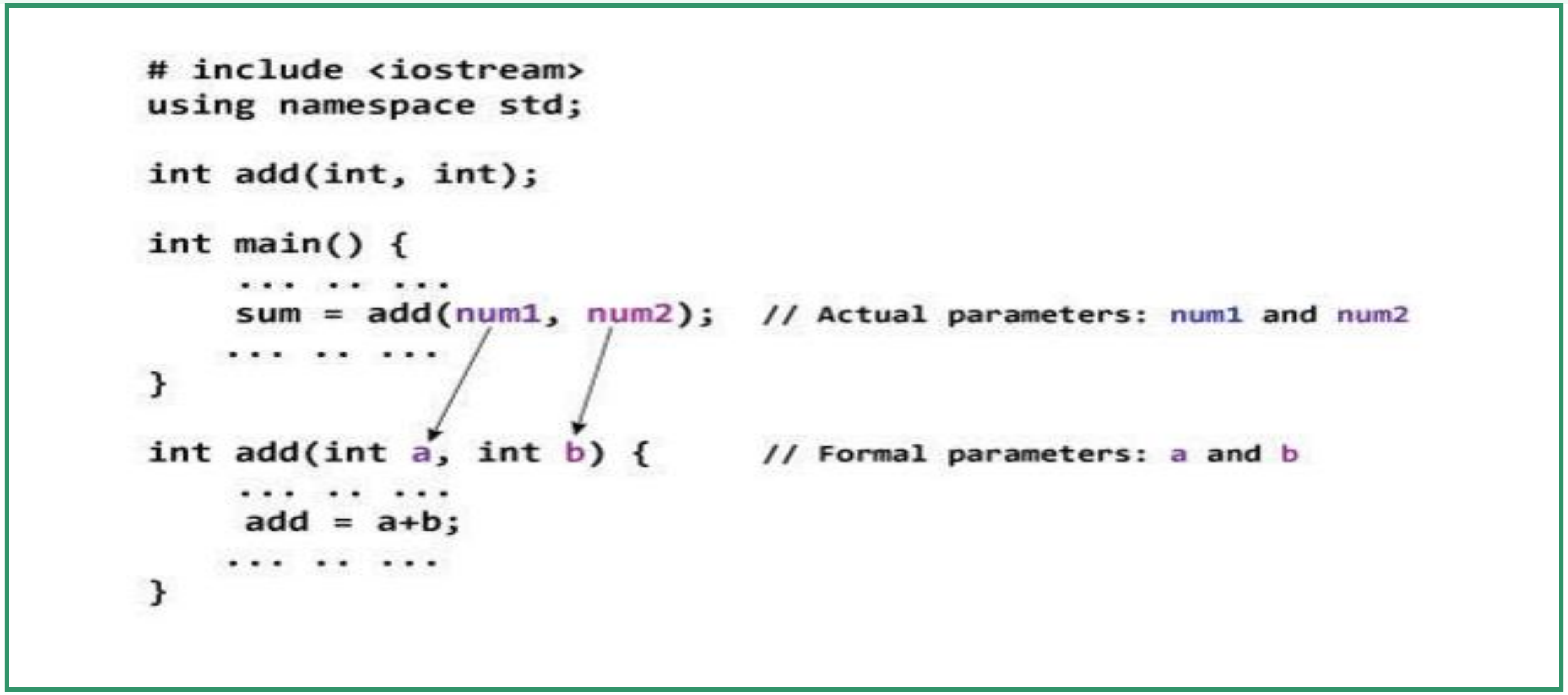
Function Definition

The function itself is referred as function definition. Function definition in the above program:

```
/* Function definition */
int add(int a,int b)
{ // Function declaratory
int add;
add = a+b;
 return add;        // Return statement
}
```

Passing Arguments to Function

In programming, argument(parameter) refers to data this is passed to function(function definition) while calling function.

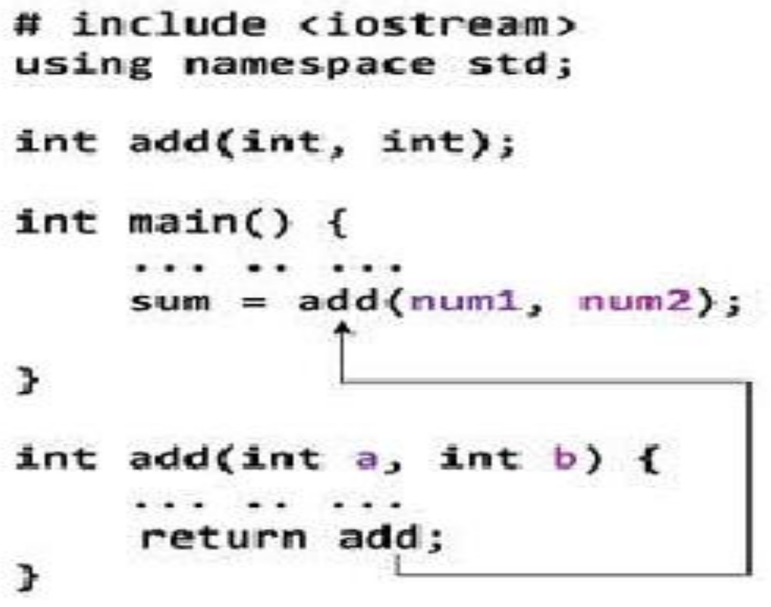


```
# include <iostream>
using namespace std;

int add(int, int);

int main() {
     ... .. ...
     sum = add(num1, num2);   // Actual parameters: num1 and num2
    ... .. ...
}

int add(int a, int b) {      // Formal parameters: a and b
     ... .. ...
     add = a+b;
    ... .. ...
}
```

Return Statement

A function can return single value to the calling program using return statement.

In the above program, the value of *add* is returned from user-defined function to the calling program using statement below:



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

int add(int, int);

int main() {
    ... .. ...
    sum = add(num1, num2);

}

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

Function Types

- For better understanding of arguments and return in functions, user-defined functions can be categorized as:

  A. **Function with no argument and no return value**

  B. **Function with no argument but return value**

  C. **Function with argument but no return value**

  D. **Function with argument and return value**

Let's consider an example to find whether a number is prime or not by making user defined function in above 4 categories .

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

```
#include <iostream>
using namespace std;
int prime();
int main() {
 int num, i;    bool flag = 0;
num = prime(); /* No argument is passed to 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; // return a value to calling function
}
```

```
#include <iostream>
using namespace std;
void prime(int n);
int main() {
int num;
 cout<<"Enter a positive integer to check: ";
 cin>>num;
prime(num); // Argument num is passed to function.
 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.";
} }
```

```
#include <iostream>
using namespace std;
int prime(int n);
 int main() {
 int num,
 flag = 0;
cout<<"Enter positive enter to check: ";
cin>>num;
flag = prime(num); /* Argument num is passed to check() function. */
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; }
```

Function Overloading

In some programming languages, **function overloading** or **method overloading** is the ability to create multiple methods of the same name with different implementations.

In C++ programming, two functions can have same identifier(name) if either number of arguments or type of arguments passed to functions are different. These types of functions having similar name are called overloaded functions.

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

Function Overloading(Con’t)

We can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. We can not overload function declarations that differ only by return type.

Overloaded function may or may not have different return type but it should have different argument(either type of argument or numbers of argument passed).

Function Overloading(Con't)

Function overloading is usually used to enhance the readability of the program. If we have to perform one single operation but with different number or types of arguments, then we can simply overload the function.

```
/* Example of function overloading */

int test() { }
int test(int a){ }
int test(double a){ }
int test(int a, double b){ }
```

Function Overloading

Two functions shown below are not overloaded functions because they only have same number of arguments and arguments in both functions are of type int.

```
/* Both functions has same number of argument and same type of argument*/
/* Hence, functions mentioned below are not overloaded functions.*/
 /* Compiler shows error in this case. */
int test(int a){ }
double test(int b){ }
```

Ways to Overloading

- ❑ By changing number of Arguments.
- ❑ By having different types of argument.

By changing number of Arguments.

In this type of function overloading we define two functions with same names but different number of parameters of the same type. For example, in the below mentioned program we have made two sum() functions to return sum of two and three integers.

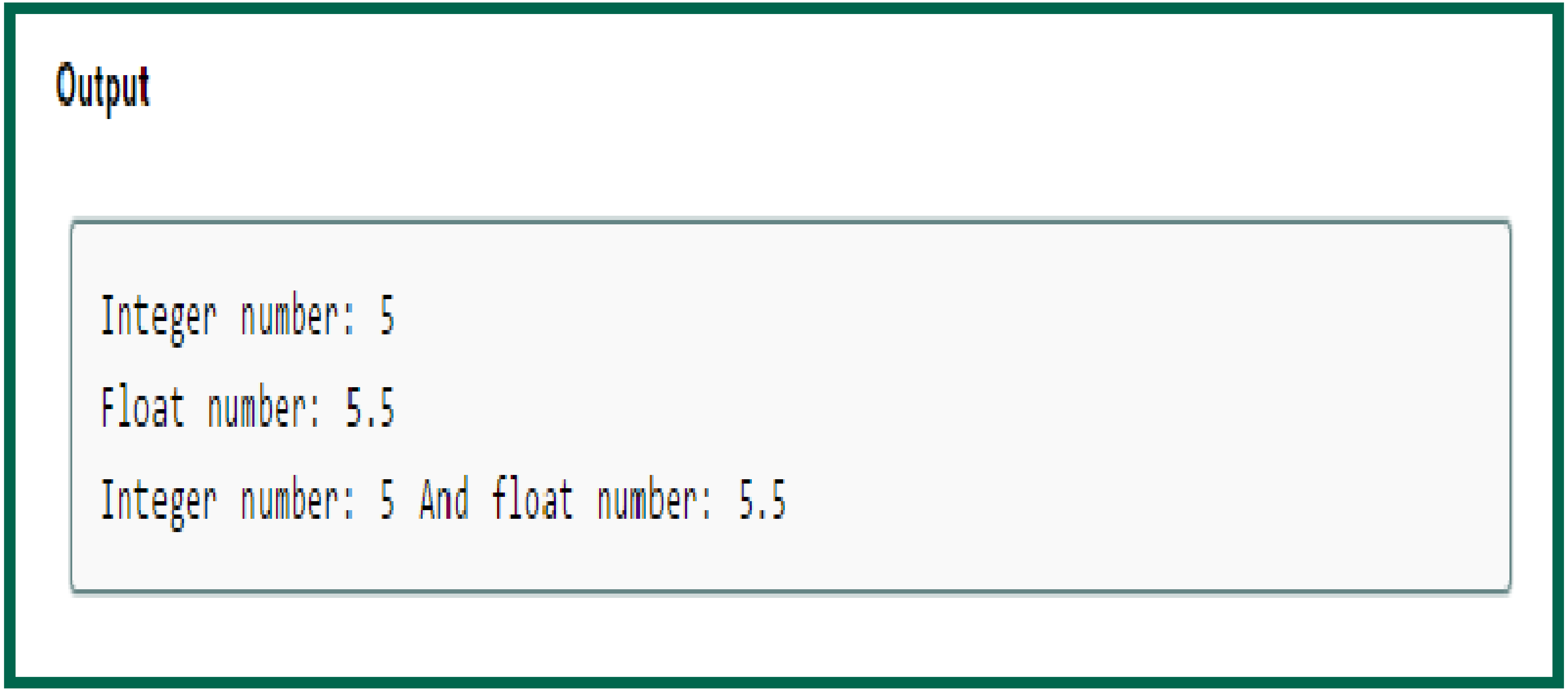
```
int sum (int x, int y)
{
 cout << x+y;
}

int sum(int x, int y, int z)
{
 cout << x+y+z;
}
```

Different data types of argument.

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different. For example in this program, we have two sum() function, first one gets two integer arguments and second one gets two double arguments.

```
int sum(int x,int y)
{ cout<< x+y; }
 double sum(double x,double y) {
 cout << x+y;
}
 int main() {
sum (10,20);
sum(10.5,20.5); }
```

```
/*Calling overloaded function test() with different argument/s.*/
#include <iostream>
using namespace std;
void test(int);
void test(float);
void test(int , float);
int main() {
int a = 5;
float b = 5.5;
 test(a);
test(b);
test(a, b);
return 0; }
void test(int var) {
cout<<"Integer number: "<<var<<endl; }
void test(float var){
cout<<"Float number: "<<var<<endl; }
void test(int var1, float var2) {
cout<<"Integer number: "<<var1; cout<<" And float number:"<<var2;
 }
```



Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 And float number: 5.5
```

Different data types of argument.

Write a C++ program that will take two different data types argument using function overloading.

C++ Default Arguments

- In C++ programming, we can provide default values for function parameters. The idea behind default argument is very simple.
- If a function is called by passing argument/s, those arguments are used by the function.

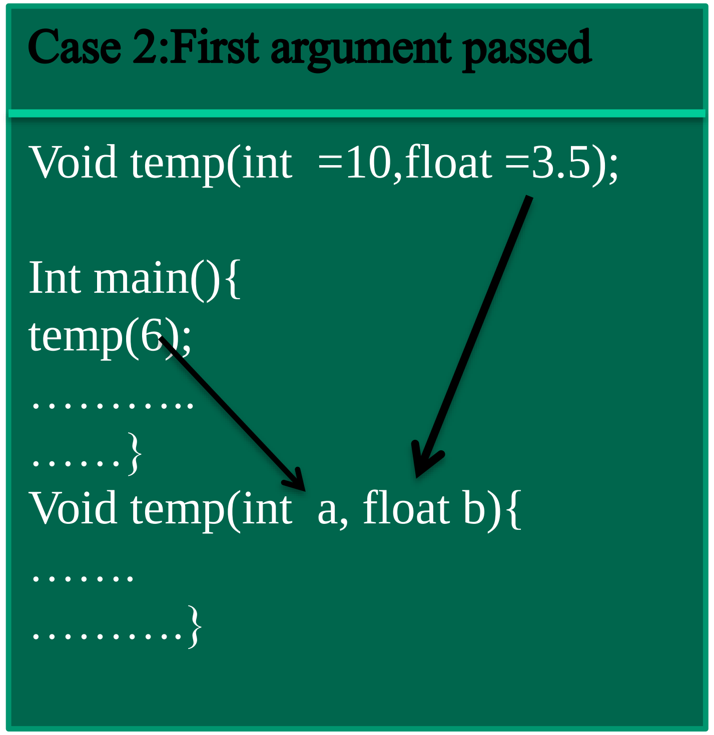
- But if all argument/s are not passed while invoking a function then, the default value passed to arguments are used.
- Default value/s are passed to argument/s in function prototype.

C++ Default Arguments

Default value/s are passed to argument/s in function prototype. Working of default argument is demonstrated in the figure below:

Case 1:No argument passed

```
void temp(int  =10,float =3.5);

int main(){
temp();
………..
……}
void temp(int  a, float b){
…….
……….}
```



Case 2:First argument passed

```
Void temp(int  =10,float =3.5);

Int main(){
temp(6);
………..
……}
Void temp(int  a, float b){
…….
……….}
```

C++ Default Arguments

Default value/s are passed to argument/s in function prototype. Working of default argument is demonstrated in the figure below:

Case 3:Both argument passed

```
Void temp(int  =10,float =3.5);

Int main(){
temp(6,4.7);
………..
……}
Void temp(int  a, float b){
…….
……….}
```

Case 4:Second argument passed

```
void temp(int  =10,float =3.5);
int main(){
temp(4.7);
………..
……}
void temp(int  a, float b){
……..}
```

Error: Missing argument must be last argument

Scope of Variables

- Every variable in C++ has a type which specifies the type of data that can be stored in a variable. For example: int, float, char etc.
- Variables and objects in C++ have another feature called storage class. Storage class specifies control two different properties: storage duration and scope.
- The variables can be divided into two main ways depending upon the storage duration and scope of variables:

  1.Local Variables

  2. Global Variables

Local Variables

- A variable defined inside a function(defined inside function body between braces) is a local variable.
- Local variables are created when the function containing local variable is called and destroyed when that function returns.
- Local variables can only be accessed from inside a function in which it exists.

Write a C++ program that will take two local variables a, b and add them in a user defined function "sum". And return the value of sum to main function.

Global Variables

- If a variable is defined outside any function, then that variable is called a global variable.
- Any part of program after global variable declaration can access global variable.
- If a global variable is defined at the beginning of the listing, global variable is visible to all functions.

C++ Global Variable Example

```
/* In this example, global variable can be accessed by all functions because it is defined at the top of
the listing.*/
#include <iostream>
using namespace std;
 int c = 12;
 void test();
 int main() {
 ++c;
cout<<c<<endl;
//Output: 13
 test();
return 0; }
 void test() {
++c;
cout<<c;
//Output: 14
}
```

Recursion

In C++, it is possible to call a function from a same function. This function is known as recursive function and this programming technique is known as recursion.

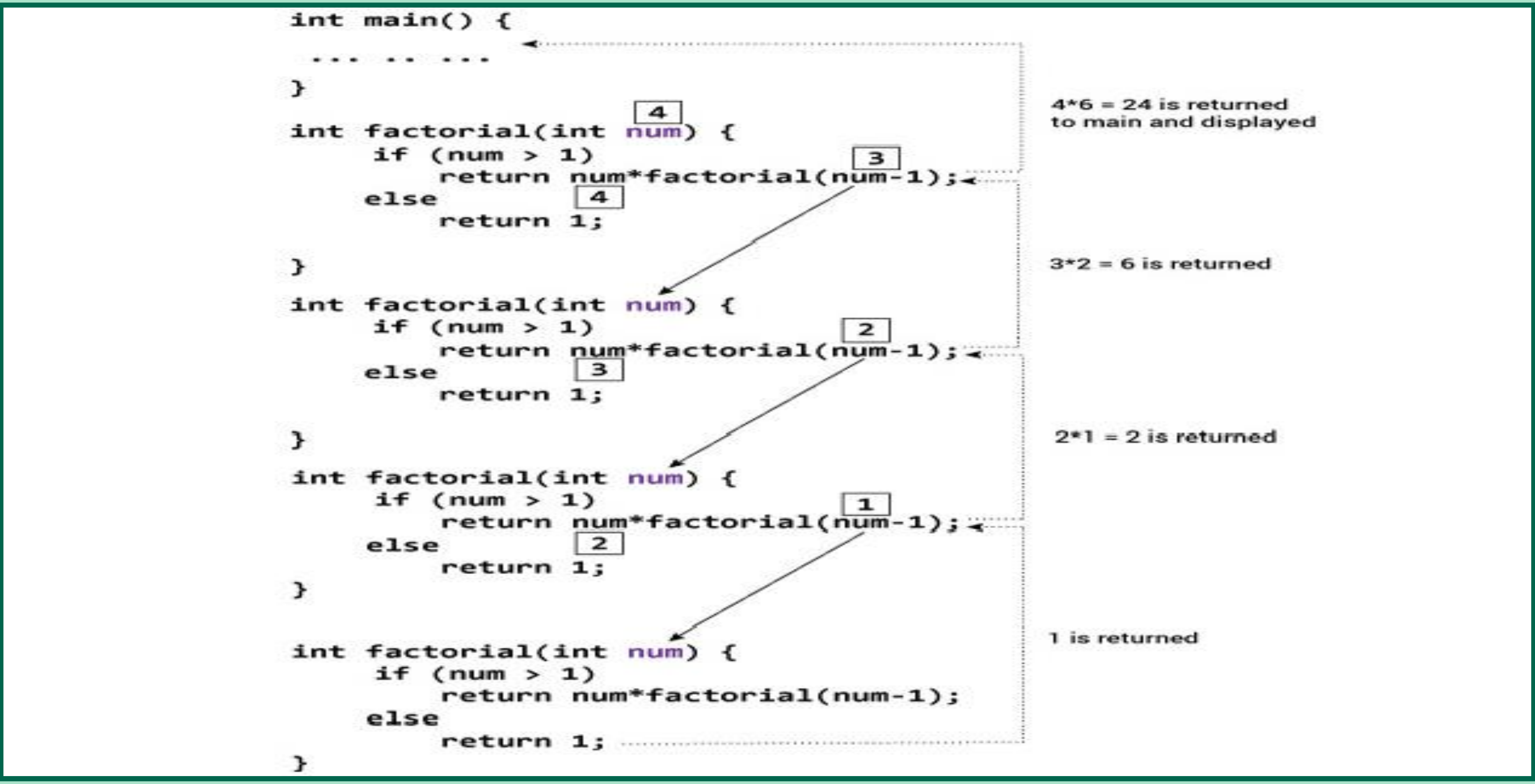
In recursion, a function calls itself but we shouldn't assume these two functions are same function. They are different functions although they have same name.

- ➢ Local variables are variables defined inside a function and has scope only inside that function.
- ➢ In recursion, a function call itself but these two functions are different functions (can imagine these functions are function1 and function 2. The local variables inside function1 and function2 are also different and can only be accessed within that function.

C++ Recursion Example

```
#include <iostream>
using namespace std;
int factorial(int);
 int main() {
int n;
 cout<<"Enter a number to find factorial: ";
 cin>>n;
 cout<<"Factorial of "<<n<<" = "<<factorial(n);
return 0; }
int factorial(int n) {
if (n>1) {
 return n*factorial(n-1);
 } else {
return 1; } }
```

C++ Recursion Example



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

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

}

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

}

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

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