INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

EEC-101
Programming with C++

Module-2:
Programming through Functional Decomposition

About Subject

- Functions
  - Parameters,
  - scope and lifetime of variables,
  - passing by value,
  - passing by reference,
  - passing arguments by constant reference
- Design of functions and their interfaces
- Recursive functions
- Function overloading and Default Arguments
- Library functions
- Matters of style (naming conventions, comments)

Functions in cmath

- Perform common mathematical calculations
  - Include the header file **<cmath>**

- Functions called by writing
  - functionName (argument);

  or
  - functionName (argument1, argument2, …);

- Example

  ```
  x =sqrt( 900.0 );
  ```
  - sqrt () function
  - The preceding statement would print 30

  All functions in math library return a **double**

Function arguments can be

- Constants
  - `sqrt( 4.0 );`
- Variables
  - `sqrt( x );`
- Expressions
  - `sqrt( sqrt( x ) ) ;`
  - `sqrt( 3 - 6x );`

Function	Description	Example
ceil(x)	rounds x to the smallest integer not less than x	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	trigonometric cosine of x (x in radians)	cos(0.0) is 1.0
exp(x)	exponential function ex	exp(1.0) is 2.71828 exp(2.0) is 7.38906
fabs(x)	absolute value of x	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
floor(x)	rounds x to the largest integer not greater than x	floor(9.2) is 9.0 floor(-9.8) is -10.0
fmod(x, y)	remainder of x/y as a floating-point number	fmod(13.657, 2.333) is 1.992
log(x)	natural logarithm of x (base e)	log(2.718282) is 1.0 log(7.389056) is 2.0
log10(x)	logarithm of x (base 10)	log10(10.0) is 1.0 log10(100.0) is 2.0
pow(x, y)	x raised to power y (xy)	pow(2.0, 7) is 128 pow(9.0, .5) is 3
sin(x)	trigonometric sine of x (x in radians)	sin(0.0) is 0
sqrt(x)	square root of x	sqrt(900.0) is 30.0
tan(x)	trigonometric tangent of x (x in radians)	tan(0.0) is 0

Functions

- A function is a block that takes input, does some specific computation, and produces output.
- The idea is to put some commonly or repeatedly done tasks together to make a function so that instead of writing the same code again and again for different inputs, we can call this function.
- In simple terms, a function is a block of code that runs only when it is called.

- Named program-pieces which can be called by the name
- Provide program reuse without repeating the whole code
- To use a function –
    1. Function Definition
    2. call the function where needed

WAP to find the sum of four different sets of numbers.

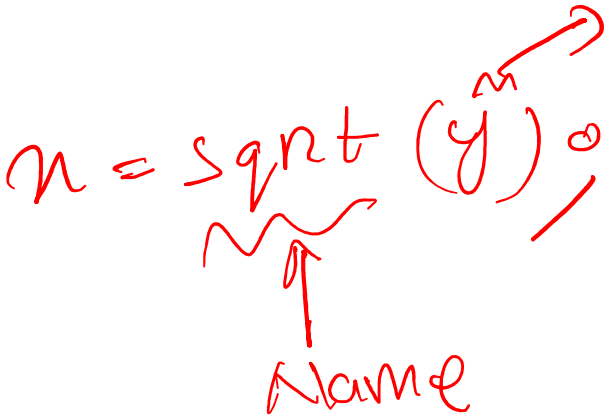
```
# include <iostream>
using namespace std;
int main ()
{
cout<< "Enter Two integers";
int a,b, sum;
cin>>a>>b;
sum=a+b;
cout<< "sum of "<<a<<" and "<<b<<" ="<<sum<<endl;

cout<< "Enter Two integers";
cin>>a>>b;
sum=a+b;
cout<< "sum of "<<a<<" and "<<b<<" ="<<sum<<endl ;

cout<< "Enter Two integers";
cin>>a>>b;
sum=a+b;
cout<< "sum of "<<a<<" and "<<b<<" ="<<sum<<endl ;


return 0;
}
```

Same program using function

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

void sum()
{
cout<< "Enter Two integers";
int a,b, sum;
cin>>a>>b;
sum=a+b;
cout<< "sum of "<<a<<" and "<<b<<" ="<<sum<<endl;
}
int main ()
{

sum();
sum();
sum();
return 0;
}
```

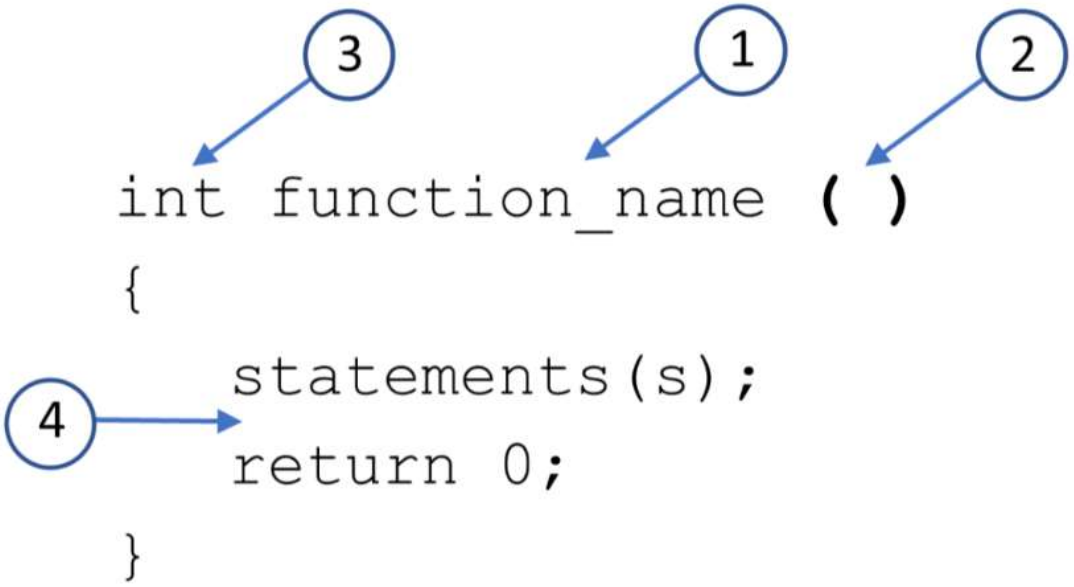
Syntax:

- name
  - the name of the function
  - same rules as for variables
  - should be meaningful
  - usually a verb or verb phrase
- parameter list
  - the variables passed into the function
  - their types must be specified
- return type
  - the type of the data that is returned from the function
- body
  - the statements that are executed when the function is called
  - in curly braces {}



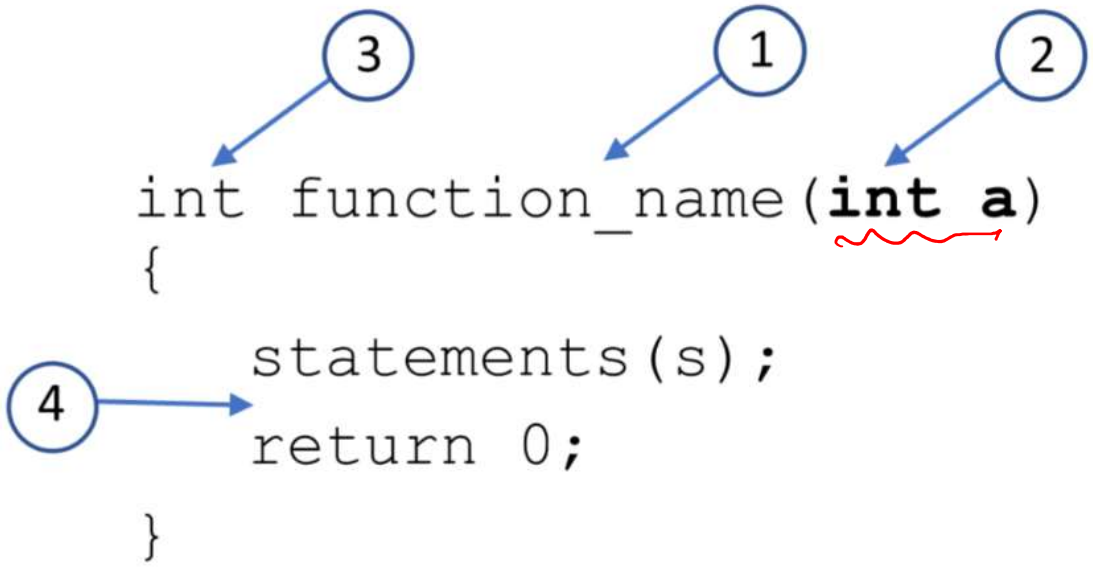
Defining Function

Example with no parameters



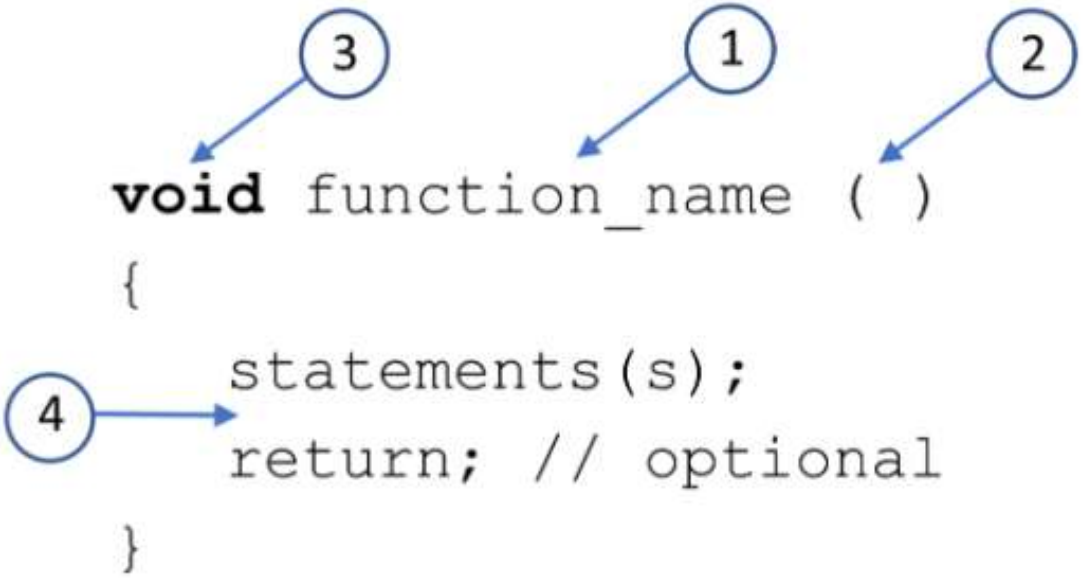
1. Name
2. Parameters
3. Return type
4. Body

Example with 1 parameter



1. Name
2. Parameters
3. Return type
4. Body

Example with no return type (void)



1. Name
2. Parameters
3. No return type
4. Body

- WAP to find max of two numbers using function. Fill in the blanks.

```
#include <iostream>
using namespace std;
// Following function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers as integer
int      (      x, int y)
{
    if (x > y)
        return x;
    else
        
}
// main function that doesn't receive any parameter and
// returns integer
int main()
{
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    m = max(a, b);

    cout << "m is " << m;
    return 0;
}
```

- Name
- Parameter List
- Return Type
- Body

Example with multiple parameters

```
void function_name(int a, std::string b)
{
   statements(s);
   return; // optional
}
```

A function with no return type and no parameters

```
void say_hello () {
    cout << "Hello" << endl;
}
```

Calling A Function

In order to transfer control to a user-defined function, we need to call it. Functions are called using their names followed by round brackets. Their arguments are passed inside the brackets.

```
void say_hello () {
    cout << "Hello" << endl;
}

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

```
void say_hello () {
    cout << "Hello" << endl;
}

int main() {
    for (int i{1} i<=10; ++i)
        say_hello();
    return 0;
}
```

Calling Functions

- Functions can call other functions
- Compiler must know the function details **BEFORE** it is called!

```
int main() {
    say_hello();  // called BEFORE it is defined ERROR
    return 0;
}

void say_hello ()
{
    cout << "Hello" << endl;
}
```

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

void say_hello(); // prototype
void say_world();
int main ()
{
    cout<<"In main \n";
    say_hello();
    cout<<"Exit main\n";
    return 0;
}
void say_hello()
{
    cout<<"In hello\n";
    cout<<"Hello ";
    say_world();
    cout<<"Exit hello\n";
}
```

```cpp
void say_world()
{
    cout<<"World !\n";
    cout<<"Exit world\n";
}
```

```
In main
In hello
Hello World !
Exit world
Exit hello
Exit main
```

Function Prototypes

- **The compiler must 'know' about a function before it is used**
  - Define functions before calling them
    - OK for small programs
    - Not a practical solution for larger programs
  - Use function prototypes
    - Tells the compiler what it needs to know without a full function definition
    - Also called forward declarations
    - Placed at the beginning of the program

Function Prototype:

- It describes the function ***interface*** to the compiler. Introduces three things –
  - data-type of the return value
  - function name
  - number and data type of parameters taken by the function
  - the order in which parameters would be provided

```
int function_name();  // prototype
```

```
int function_name()
{
    statements(s);
    return 0;
}
```

```
int function_name(int);   // prototype
                          // or
int function_name(int a); // prototype
```

```
int function_name(int a) {
    statements(s);
    return 0;
}
```

Function Parameters

- When we call a function we can pass in data to that function
- In the function call they are called arguments
- In the function definition they are called parameters
- They must match in number, order, and in type

```
int add_numbers(int, int);          // prototype

int main() {
    int result {0};
    result = add_numbers(100,200);  // call
    return 0;
}

int add_numbers(int a, int b) { // definition
    return a + b;
}
```

```
//*************************************************************
// Triangle program
// This program uses the IsTriangle function
//*************************************************************
#include <iostream>
#include <cmath>       // For fabs()

using namespace std;

bool IsTriangle(float, float, float);

int main()
{
  float angleA;     // Three potential angles of a triangle
  float angleB;
  float angleC;
  cout << "Triangle testing program; "
       << "a negative first angle ends the processing." << endl;
  cout << "Enter 3 angles: ";
  cin >> angleA;
  while (angleA >= 0)
  {
    cin >> angleB >> angleC;
    if (IsTriangle(angleA, angleB, angleC))
      cout << "The 3 angles form a valid triangle." << endl;
    else
      cout << "The 3 angles do not form a triangle." << endl;
    cout << "Enter 3 angles: ";
    cin >> angleA;
  }

  return 0;
}
//*************************************************************
bool IsTriangle(float angle1, float angle2, float angle3)
{
  return (fabs(angle1 + angle2 + angle3 - 180.0) < 0.00000001);
}
```

Parameter passing

- Parameters provide a communication channel between
  - function, and calling environment.
- The calling environment is another function – main or some other function.
- Calling environment can be same function also – recursive call

- The channel for a parameter can be –
  - One way (input only), or
  - Two way (both input and Output)

- There are three methods of passing parameters
  - pass by value
  - pass by reference
  - pass by constant reference

Pass by value:

- Provides a one way channel (input only) from the calling environment to the function
- a *copy* of the value is passed at the time of call
- The value passed is used at all occurrences of the formal parameter inside the function body.
- the actual parameter can be in the form of an expression, which is evaluated before passing.
- any changes done to the parameter are in the copy, hence not visible in the calling environment. The parameter is read only (input only).
- because a copy is passed, the method is not efficient for passing big data.

Pass by Value

- When you pass data into a function it is passed-by-value
- A copy of the data is passed to the function
- Whatever changes you make to the parameter in the function does NOT affect the argument that was passed in.

- Formal vs. Actual parameters
  - Formal parameters – the parameters defined in the function header
  - Actual parameters – the parameter used in the function call, the arguments

```
void param_test(int formal) {  // formal is a copy of actual
    cout << formal << endl;    / 50
    formal = 100;              // only changes the local copy
    cout << formal << endl;   // 100
}

int main() {
    int actual {50};
    cout << actual << endl;   // 50
    param_test(actual);        // pass in 50 to param_test
    cout << actual << endl;   // 50   - did not change
    return 0
}
```

```
#include<iostream>
using namespace std;
// Demonstrating difference between actual and formal arguments
int addTwoInts(int, int);  // Prototype
int main()  // main function
{
  int n1 = 10, n2 = 20, sum;
  /* n1 and n2 are actual arguments. They are the source of data.
   Caller program supplies the data to called  function in form of actual arguments. */

  sum = addTwoInts(n1, n2); // function call

  cout<< "Sum of " <<n1<< " and "<<n2<<" is: " <<sum;
}
/* a and b are formal parameters.
They receive the values from actual arguments when this function is called. */
int addTwoInts(int a, int b)
{
  return (a + b);
}
```

Return Statement

- If a function returns a value then it must use a `return` statement that returns a value
- If a function does not return a value (`void`) then the `return` statement is optional
- `return` statement can occur anywhere in the body of the function
- `return` statement immediately exits the function
- We can have multiple `return` statements in a function
  - Avoid many return statements in a function
- The return value is the result of the function call

Function types

- **Value returning**:
  - Such functions return a value when called.
  - Call for such a function will usually appear on the RHS of an assignment statement, in a cout statement, or as part of an expression.
  - The calling environment receives the returned value.
  - There will be a return statement in the function body.

- **void type**:
  - They do not return value but rather produce an effect.
  - Function call can be a stand-alone statement.
  - No need for a return statement in the body.

Default Argument Values

- When a function is called, all arguments must be supplied
- Sometimes some of the arguments have the same values most of the time
- We can tell the compiler to use default values if the arguments are not supplied
- Default values can be in the prototype or definition, not both
  - best practice – in the prototype
  - must appear at the tail end of the parameter list
- Can have multiple default values
  - must appear consecutively at the tail end of the parameter list

Default Argument Values

Example – no default arguments

```
double calc_cost(double base_cost, double tax_rate);

double calc_cost(double base_cost, double tax_rate) {
    return base_cost += (base_cost * tax_rate);
}

int main() {
    double cost {0};
    cost = calc_cost(100.0, 0.06);
    return 0;
}
```

Example – single default argument

```
double calc_cost(double base_cost, double tax_rate = 0.06);

double calc_cost(double base_cost, double tax_rate) {
   return base_cost += (base_cost * tax_rate);
}

int main() {
   double cost {0};
   cost = calc_cost(200.0);   // will use the default tax
   cost = calc_cost (100.0, 0.08);      // will use 0.08 not the defauly
   return 0;
}
```

Example – multiple default arguments

```
double calc_cost(double base_cost, double tax_rate = 0.06, double shipping = 3.50);

double calc_cost(double base_cost, double tax_rate, double shipping) {
   return base_cost += (base_cost * tax_rate) + shipping;
}

int main() {
   double cost {0};
   cost = calc_cost (100.0, 0.08, 4.25);  // will use no defaults
   cost = calc_cost(100.0, 0.08);     // will use default shipping
   cost = calc_cost(200.0);      // will use default tax and shipping
return 0;
}
```

- When we mention a default value (preset value) for a parameter while declaring the function, it is said to be as default argument.

- In this case, even if we make a call to the function without passing any value for that parameter, the function will take the default value specified.

- By setting default argument, we are also overloading the function. Default arguments also allow us to use the same function in different situations just like function overloading.

Rules for using Default Arguments

1.Only the last argument must be given default value. One cannot have a default argument followed by non-default argument.

```
sum (int x,int y);
sum (int x,int y=0);
sum (int x=0,int y); // This is Incorrect
```

2.If you default an argument, then we will have to default all the subsequent arguments after that.

```
sum (int x,int y=0);
sum (int x,int y=0,int z); // This is incorrect
sum (int x,int y=10,int z=10); // Correct
```

3.We can give any value a default value to argument, compatible with its datatype.

- During calling of function, arguments from calling function to called function are copied from left to right
- If the user does not supply a value for this parameter, the default value will be used.
- If the user does supply a value for the default parameter, the user-supplied value is used instead of the default value.
- ***Rule: If the function has a forward declaration (prototype), put the default parameters there. Otherwise, put them on the function definition (but not on both)***

Examples

Example:

1. int volume (int length = 1, int width = 1, int height = 1); // valid
2. int volume (int length = 1, int width, int height = 1); // not allowed
3. int volume (int length, int width = 1, int height = 1); // valid

Alternate way for prototypes

1. int volume (int = 1, int = 1, int = 1);
2. int volume (int, int = 1, int = 1);

Some calls to the function

(a) volume() //valid for declaration no.1 but not 3

(b) volume(5) //valid for both, width = height = 1

(c) volume(5,7) //valid for both, height = 1

Scope of Variables

- **Scope of a Variable** defines the part of code where the variable can be accessed or modified. It helps in organizing code by limiting where variables are available and preventing unintended changes or conflicts.

- Variables can be broadly classified as
  - Local Variables
  - Global Variables

Local Variables

- Variables defined within a function or block are said to be local to those functions.
- Local variables do not exist outside the block or function in which they are declared, i.e. they **can not** be accessed or used outside that block.
- Scope of the local variables is limited to the function in which these variables are declared.

For example, if they are declared at the beginning of the body of a function (like in function *main*) their scope is between its declaration point and the end of that function

This means that if another function existed in addition to *main*, the local variables declared in *main* could not be accessed from the other function and vice versa.

Example

```
#include <iostream>
    int sum(int,int);
    int main( )
     {
        int b;
        int s=5,u=6;
        b= sum(s,u);       //function call
        cout<<"\n The Output is:"<<b;
        system("pause");
        return 0;
     }
     int sum(int x, int y) //function definition
     {
        int z;
        z=x+y;
        return(z);
     }
```

- Variables x, y, z are accessible only inside the function sum( ).
- Their scope is limited only to the function sum( ) and not outside the function.
- Thus the variables x, y, z are local to the function sum.

- Similarly one would not be able to access variable b inside the function sum as such. This is because variable b is local to function main.

Global Variables

- All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code
- A **global variable** is a variable declared in the main body of the source code, outside all functions
- Global variables can be referred from anywhere in the code, even inside functions.

Example

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

int x= 10;
int main()
{
    int x =20;
    {
    int x = 30;
    cout << "x= "<< x<< endl;
    }
cout << "x= "<< x<< endl;
cout << "x= "<<:: x<< endl;
return 0;
}
```

```
x= 30
x= 20
x= 10

Process returned 0 (0x0)   ex
Press any key to continue.
```

```
/*******************************************************
Global and local variables.
*******************************************************/

#include <iostream>
using namespace std;

void SomeFunc(float);        // Function Prototype

const int A = 17;               // Global constant
int b;                          // Global variable
int c;                          // Another global variable

int main()
{
   b = 4;                      // Assignment to global b
   c = 6;                      // Assignment to global c
   SomeFunc(42.8);
   cin.ignore();
   cin.get();
   return 0;
}
 /*******************************************************/
void SomeFunc(float c)        // Prevents access to global c
{
  float b;                                  // Prevents access to global b
  b = 2.3;                                  // Assignment to local b
  cout << " A = " << A<<endl;               // Output global A (17)
  cout << " b = " << b<<endl;               // Output local b (2.3)
  cout << " c = " << c<<endl;               // Output local c (42.8)
  cout << " global b = " <<:: b<<endl;      // Output global b
  cout << " global c = " <<:: c;<<endl      // Output global b
}
```

```
A = 17
b = 2.3
c = 42.8
global b = 4
global c = 6


Process returned 0 (0x0)   execution time : 5.538 s
Press any key to continue.
```

Terminology

- Let us assume that a function *B()* is called from another function *A()*. In this case, *A* is called the ***"caller function"*** and *B* is called the ***"called function"***.
- The arguments which *A* sends to *B* are called *actual arguments* and the parameters of *B* are called *formal arguments*.
- **Terminology**
- **Formal Parameter:** A variable and its type as it appears 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.

Parameter Passing

- Parameters provide a communication channel between
  - function, and calling environment.
- The calling environment is another function – main or some other function.
- Calling environment can be the same function also – recursive call

- The channel for a parameter can be –
  - One way (input only), or

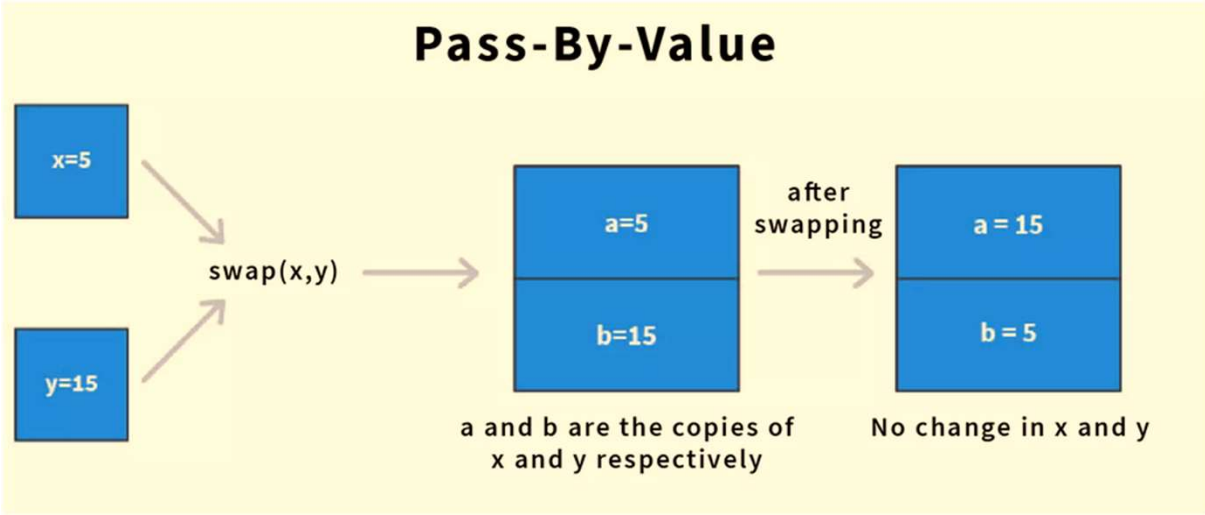
  - Two way (both input and Output)
- There are three methods of passing parameters
  1. pass by value
  2. pass by reference
  3. pass by constant reference

Pass by Value

- Provides a one way channel (input only) from the calling environment to the function
- A *copy* of the value is passed at the time of call
- The value passed is used at all occurrences of the formal parameter inside the function body.
- The actual parameter can be in the form of an expression, which is evaluated before passing.
- Any changes done to the parameter are in the copy, hence not visible in the calling environment. The parameter is read only (input only).
- Because a copy is passed, the method is inefficient for passing big data.

Example

```
#include <iostream>
// function to swap two values
void swap(int a, int b)
 {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
// main function
int main()
{
  int x = 5, y = 15;
  // pass by values
  swap(x, y);
  std::cout << "x:" << x << ", y:" << y;
  return 0;
}
```




```
x:5, y:15
Process returned 0 (0x0)   execution time : 0.059 s
Press any key to continue.
```

Pass by reference:

- Provides a two-way channel (input and output)
- Instead of passing a copy of data, the address of data is passed.
- Any changes done to reference parameters are seen by the calling environment also. Thus reference parameters are read-write type (both input and output).
- Thus, the function gets an additional route for sending back results to the calling environment.
- The method is efficient for big data also because only the address is passed, and no copying is done.
- Sometimes a reference parameter is put in a function for only receiving a result – in that case it is a one way channel for output only (no input is planned through it).

Reference Variable

- When a variable is declared as a reference, it becomes an alternative name for an existing variable. A *shortcut*.
- A variable can be declared as a reference by putting '&' in the declaration.

- Reference variable is a type of variable that can act as a reference to another variable. '&' is used for signifying the address of a variable or any memory.
- Variables associated with reference variables can be accessed either by their name or by the reference variable associated with them.

```cpp
#include <iostream>
using namespace std;
int main()
{

int dhoni;
int &mahi = dhoni; // mahi is the same variable as dhoni
dhoni = 7;
cout << "dhoni's  number is " << dhoni << endl;
cout << "mahi's  number is " << mahi << endl;
dhoni++;
cout << "dhoni's  number is " << dhoni << endl;
cout << "mahi's  number is " << mahi << endl;
mahi=mahi+10;
cout << "dhoni's  number is " << dhoni << endl;
cout << "mahi's  number is " << mahi << endl;
return 0;
}
```

```
dhoni's  number is 7
mahi's  number is 7
dhoni's  number is 8
mahi's  number is 8
dhoni's  number is 18
mahi's  number is 18

Process returned 0 (0x0)   execution time : 0.056 s
Press any key to continue.
```

Difference between PBV and PBR

- Pass by Value

```
void add10( int x) {
  x = x+10;
}
…
add10(counter);
```

Effect: x is a copy of counter

- Pass by Reference

```
void add10( int &x) {
  x = x+10;
}
…
add10(counter);
```

Effect: x is a reference(shortcut) to counter

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

void Test_Function(int &var)
{
var = 3299;
}

int main()

{ int a;
a = 50;
Test_Function(a);

cout << a << endl;

return 0;  }
```

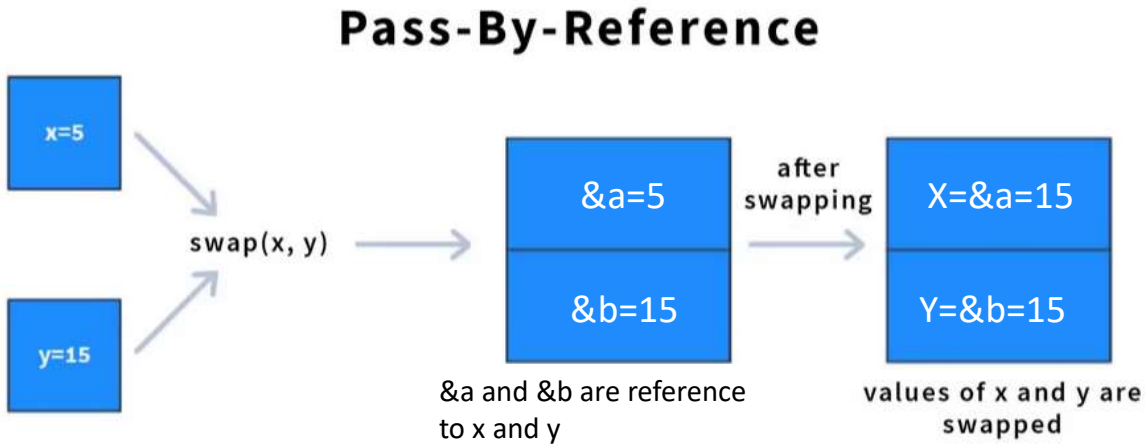
```
3299

--------------------------------
Process exited after 0.04712 seconds with return
 value 0
Press any key to continue . . .
```

Result : It will print the number 3299, because we called Test_Function.
However it would just print 50, if

1. Didn't call TestFunction or
2. Did not put the & in front of var - (it will become passing by value)

Example: Pass by reference

```
#include <iostream>
// function to swap values
void swap(int &a, int &b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
// main function
int main() {
   int x = 5, y = 15;
   //pass by reference
   swap(x, y);
   std::cout << "x:" << x << ", y:" << y;
   return 0;
}
```



Example:3

```
1   // passing parameters by reference
2   #include <iostream>
3   using namespace std;
4   void duplicate (int &a, int &b, int &c)
5   {
6    a*=2;
7    b*=2;
8    c*=2;
9    cout << "a=" << a<< " b=" << b << " c=" << c<<endl;
10  }
11  int main ()
12  {
13    int x=1, y=3, z=7;
14    duplicate (x, y, z);
15    cout << "x=" << x << ", y=" << y << ", z=" << z<<endl;
16    system("pause");
17   return 0;
18  }
19
```

```
a=2 b=6 c=14
x=2, y=6, z=14
Press any key to continue . . .
```

Advantages and Disadvantages

- References allow a function to change the value of the argument, which is useful.
- Because a copy of the argument is not made, pass by reference is fast, even when used with large structs or classes.
- References can be used to return multiple values from a function (via out parameters).
- References must be initialized, so there's no worry about null values.

Disadvantages of passing by reference:

- It's impossible to tell from the function call whether the argument may change. An argument passed by value and passed by reference looks the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.

Pass by Constant Reference

- a one-way channel (input only)
- a const reference cannot be changed, hence no output is possible
- acts like pass-by-value, but copying of data is avoided because the address is passed – no inefficiency for large data
- It can be hard to tell whether an argument passed by non-const reference is meant to be input, output, or both. Judicious use of const and a naming suffix for out variables can help.
- Gives the safety of pass-by-value and efficiency of pass-by-reference

Example

```
#include <iostream>
using namespace std;
//void tryAltering(int& byRefVal)    // uncomment this to show pass by reference
//void tryAltering(const int& byRefVal) // uncomment this to show constant reference
{
    cout << "Before: " << byRefVal<<endl;
    int y=byRefVal+10;
    byRefVal++; // try incrementing...
    // error - you cannot alter constant reference
    cout <<"y="<< y; //can use the data
    cout << "After: " << byRefVal<<endl;
}
int main(void)
{
    int somenumber = 100;
    tryAltering(somenumber);
    system("pause");
    return 0;
}
```

Advantages

- Gives the safety of pass-by-value and efficiency of pass-by-reference

Function Overloading

- A function having several prototype declarations and correspondingly several definitions under same name
- Each version of an overloaded function is distinguishable through its different set of parameters
- The signature of an overloaded version consists of the number and type of parameters. Each overloaded version should have distinguishable signature
- Overloading allows functions performing similar tasks but operating on different data-types to have same name
- At the time of call, the applicable definition is chosen on the basis of the number, type, and order of arguments

Overloading Functions

- We can have functions that have different parameter lists but have the same name
- Abstraction mechanism since we can just think 'print' for example
- A type of polymorphism
  - We can have the same name work with different data types to execute similar behavior
- The compiler must be able to tell the functions apart based on the parameter lists and argument supplied

Function Signature

A function is overloaded when same name is given to a different function.

However, the two functions with the same name will differ at least in one of the following.

a) The number of parameters
b) The data type of parameters
c) The order of appearance

These three together are referred to as the **function signature**.

Example

```
int add_numbers(int, int);     // add ints
double add_numbers(double, double);  // add doubles

int main() {
    cout << add_numbers(10,20) << endl;     // integer
    cout << add_numbers(10.0, 20.0) << endl; // double
 return 0;
}
int add_numbers(int a, int b) {
   return a + b;
}

double add_numbers(double a, double b) {
   return a + b;
}
```

```
void display(int n);
void display(double d);
void display(std::string s);
void display(std::string s, std::string t);
void display(std::vector<int> v);
void display(std::vector<std::string> v);
```

Return type is not considered

```
int    get_value();
double get_value();

// Error

cout << get_value() << endl; // which one?
```

Example

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

int operate (int a, int b)
{
return (a*b);
}
float operate (float a, float b)
 {
return (a/b);
}
int main ()
{
int x=5,y=2;
float n=5.0,m=2.0;
cout << operate (x,y);
cout << "\n";
cout << operate (n,m);
cout << "\n";
return 0; }
```

```
10
2.5

Process returned 0 (0x0)   execution time : 0.032
Press any key to continue.
```

Ambiguity

- In Function overloading, sometimes a situation can occur when the compiler is unable to choose between two correctly overloaded functions. This situation is said to be ambiguous.
- Ambiguous statements are error-generating statements and the programs containing ambiguity will not compile.

- Automatic type conversions are the main cause of ambiguity. In C++, the type of argument that is used to call the function is converted into the type of parameters defined by the function.

- bool, char, and short are implicitly converted to int.
- Float is converted to double
- int type is converted to long and double

Example

```
#include <iostream>
using namespace std;
void test(int f)
{
    cout << "Overloaded Function with "
        << "int type parameter called";
}
void test(long l)
{
    cout << "Overloaded Function with "
        << "long type parameter called";
}
int main()
{
    // Overloaded Function called
    // with float type value
    test(2.5f);
    //test((int)(2.5f)); //typecast to int to resolve ambiguity
    return 0;
}
```

The above code will throw an error because the test(2.5f) function call will look for float function if not present it is only promoted to double, but there is no function definition with double or float type of parameter.

Ex: Ambiguity

```
#include<iostream>
using namespace std;
long square (long n);
double square (double n);
int main ()
    {
     cout<<"Enter a number"<<endl;
     int n{0};

     cin>>n;
     cout<<" The square of the number is : "<<square(n);
    return (0);

    }
long square (long n){
return n*n;
}
double square (double n){
return n*n;
}
```

Ambiguity

Recursion

Recursion is a programming technique that allows the programmer to express operations in terms of themselves.

In C++, this takes the form of
a function that calls itself

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
- A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.

A Mathematical Interpretation

- Let us consider a problem that a programmer has to determine the sum of first n natural numbers; there are several ways of doing that.

Approch-1:
the simplest approach is simply to add the numbers starting from 1 to n. So the function simply looks like this,

$$f(n) = 1 + 2 + 3 + \cdots\ldots + n$$

Approch-2:
another mathematical approach is

$$f(n) = 1 \qquad n = 1$$
$$f(n) = n + f(n-1) \quad n > 1$$

F(4) = ?

4 + f(3)
4 + 3 + F(2)
4 + 3 + 2 + F(1)
4 + 3 + 2 + 1

- There is a simple difference between approach (1) and approach(2).
- In approach(2), the function **" f( ) "** is being called inside the same function itself , so this phenomenon is named recursion, and the function containing recursion is called a recursive function.

The basic idea behind recursive algorithms:

To solve a problem,

-solve a subproblem that is a smaller instance of the same problem,

-and then use the solution to that smaller instance to solve the original problem.

Properties :

- Performing the same operations multiple times with different inputs.
- In every step, we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion; otherwise, an infinite loop will occur.

Base Case

- A terminating scenario that does not use recursion to produce an answer.
- Normally, a recursive function will have a variable that performs a similar action; one that controls when the function will finally exit.
- The condition where the function will not call itself is termed the base case of the function.
- Basically, it is an if-statement that checks some variable for a condition (such as a number being less than zero or greater than some other number), and if that condition is true, it will not allow the function to call itself again.
- Or, it could check if a certain condition is true and only then allow the function to call itself.

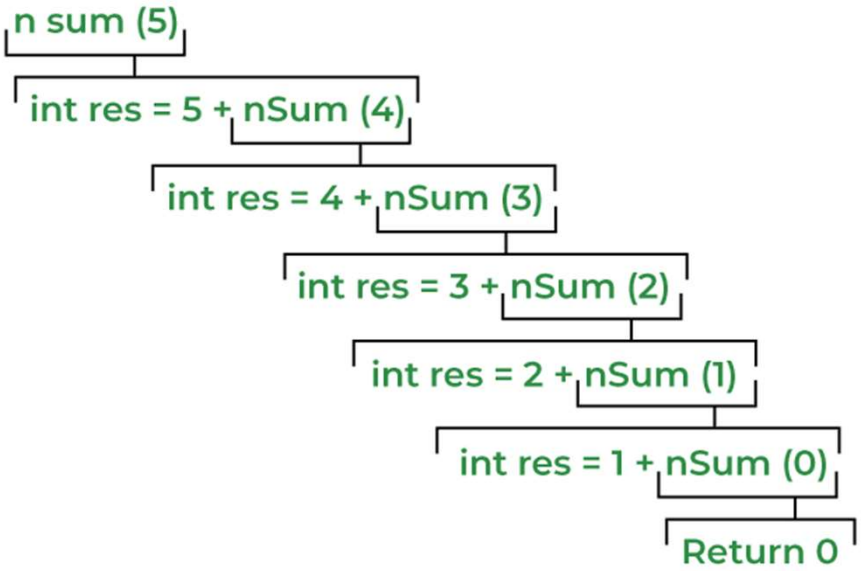
Example: sum of integers till n

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

int nsum(int a)
{
    if (a==0)
        return 0;
    else
    {
        int res=a+nsum(a-1);
        return res;
    }
}

int main ()
    {
    int n, sum;
    cout<< "Enter a number: ";
    cin>> n;
    sum=nsum(n);
    cout<<"Sum from 1 to "<<n<<" = "<<sum;
    return 0;
    }
```

Base Case

Recursion

n=5



Working of Recursion in C++

1. In the nSum() function, Recursive case is int res = n + nSum(n - 1);

2. In the example, n = 5, so as nSum(5)'s recursive case, we get int res = 5 + nSum(4);

3. In nSum(4), everything will be the same, except n = 4. Let's evaluate the recursive case for n = 4,
   int res = 4 + nSum(3);

4. Similarly, for nSum(3), nSum(2) and nSum(1)
   int res = 3 + nSum(2); // nSum(3)
   int res = 2 + nSum(1); // nSum(2)
   int res = 1 + nSum(0); // nSum(1)

Let's not evaluate nSum(0) for now.

```
int nsum(int a)
{
    if (a==0)
        return 0;
    else
    {
        int res=a+nsum(a-1);
        return res;
    }
}
```

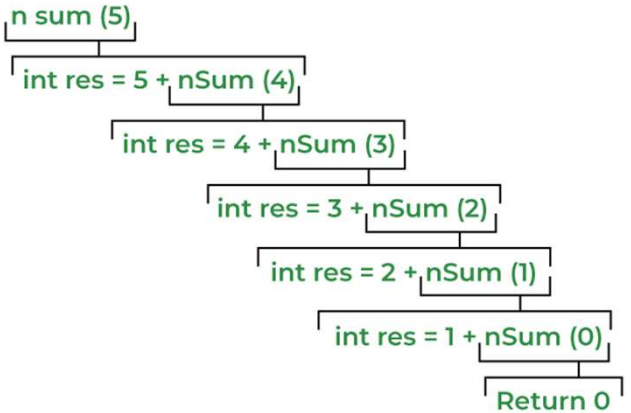
Working of Recursion in C++

```
int nsum(int a)
{
    if (a==0)
        return 0;
    else
    {
        int res=a+nsum(a-1);
        return res;
    }
}
```

5. Now recall that the return value of the nSum() function is an integer named res. So, instead of the function, we can put the value returned by these functions. As such, for nSum(5), we get int res = 5 + nSum(4);

6. Similarly, putting return values of nSum() for every n, we get

int res = 5 + 4 + 3 + 2 + 1 + nSum(0);



7. In nSum() function, the base condition is

```
if (n == 0) {
    return 0;
}
```

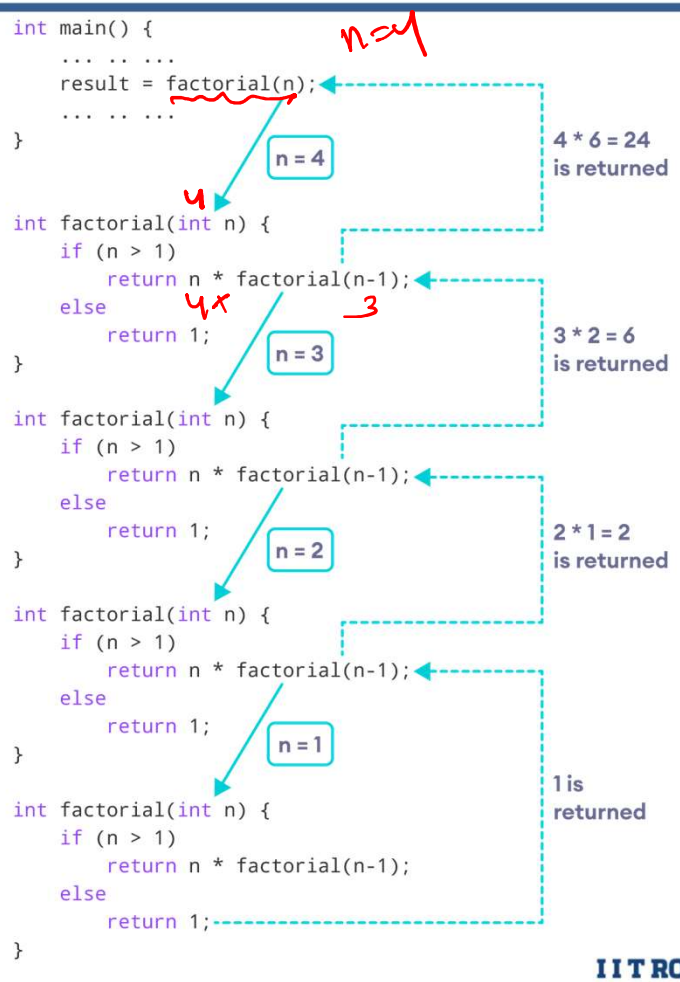
which means that when nSum(0) will return 0.
Putting this value in nSum(5)'s recursive case, we get int res = 5 + 4 + 3 + 2 + 1 + 0;

8. At this point, we can see that there are no function calls left in the recursive case. So the recursion will stop here, and the final value returned by the function will be 15, which is the sum of the first 5 natural numbers.

Example-2:Factorial of a Number Using Recursion

```cpp
#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;}
```

Draw Flowchart on your own!



Memory Management in C++ Recursion

- Like all other functions, the recursive function's data is stored in the stack memory in the form of a stack frame. This stack frame is deleted once the function returns some value.
- In recursion,
  - The function call is made before returning the value, so the stack frame for the progressive recursive calls is stored on top of existing stack frames in the stack memory.
  - When the topmost function copy returns some value, its stack frame is destroyed and the control comes to the function just before that particular copy after the point where the recursive call was made for the top copy.
  - The compiler maintains an instruction pointer to track where to return after the function execution.

Memory used by a program is typically divided into four different areas:

- The code area: where the compiled program sits in memory.
- The globals area: where global variables are stored.
- The heap: where dynamically allocated variables are allocated from.
- The stack: where parameters and local variables are allocated from.

The Stack

- The Stack is an special area of memory in which temporary variables are stored.
- The Stack acts on the LIFO (Last In First Out) principle, which is the same principle involved in, say, the stacking of cardboard boxes one atop the other, where the topmost box, which was the last box stacked (Last In), will the the first to be removed (First Out). Thus, if the values 9,3,2,4 are stored (Pushed) on the Stack, they will be retrieved (Popped) in the order 4,2,3,9.

The stack in action

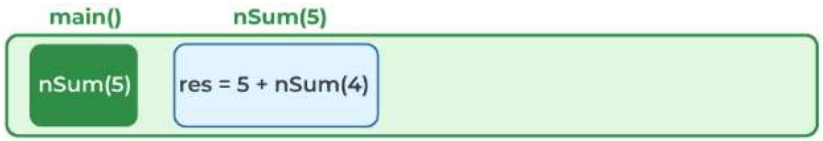
- Because parameters and local variables essentially belong to a function, we really only need to consider what happens on the stack when we call a function.
- Sequence of steps that takes place when a function is called:
- The address of the instruction beyond the function call is pushed onto the stack. This is how the CPU remembers where to go after the function returns.
- Room is made on the stack for the function's return type. This is just a placeholder for now.
- The CPU jumps to the function's code.
- The current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered "local" to the function.
- All function arguments are placed on the stack.
- The instructions inside of the function begin executing.
- Local variables are pushed onto the stack as they are defined.

When the function terminates, the following steps happen:

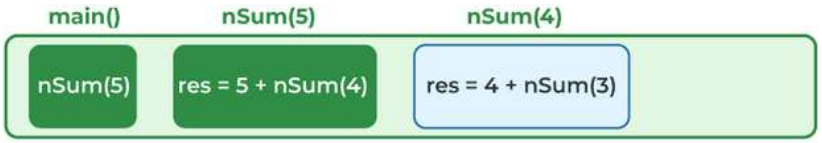
- The function's return value is copied into the placeholder that was put on the stack for this purpose.
- Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.
- The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn't assigned to anything, no assignment takes place, and the value is lost.
- The address of the next instruction to execute is popped off the stack, and the CPU resumes execution at that instruction.
- Typically, it is not important to know all the details about how the call stack works.
- However, understanding that functions are effectively pushed on the stack when they are called and popped off when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.

memory management of nSum(5)

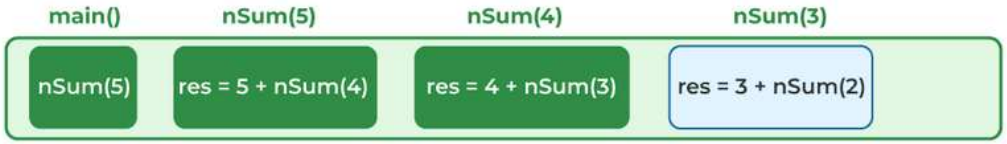
Step 1: When nSum() is called from the main() function with 5 as an argument, **a stack frame for nSum(5)** is created.



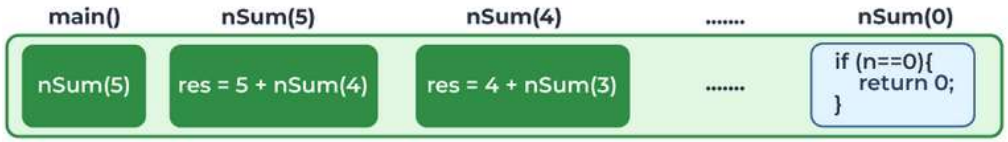
Step 2: While executing nSum(5), a **recursive call** is encountered as **nSum(4)**. The compiler will now create a new stack frame on top of the nSum(5)'s stack frame and maintain an instruction pointer at the statement where **nSum(4)** was encountered.



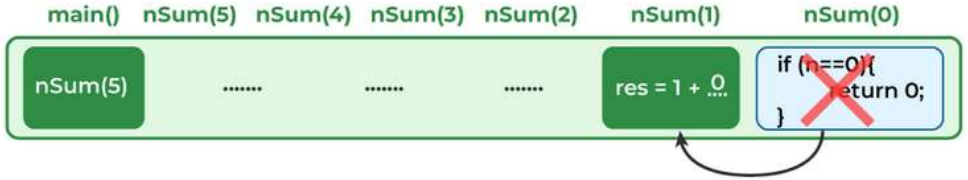
- **Step 3:** The execution of nSum(4) will start, but just like the previous function, we encounter another recursive call as nSum(3). The compiler will again follow the same steps and maintain another instruction pointer and stack frame for nSum(3).



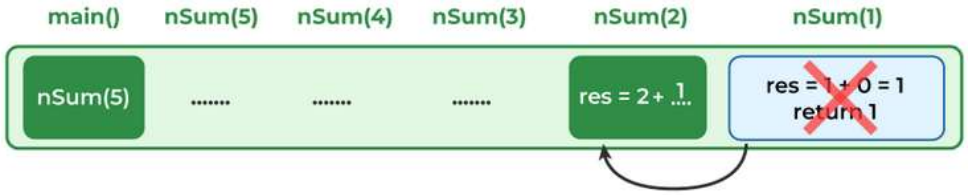
- **Step 4:** The same thing will happen with nSum(3), nSum(2), and nSum(1)'s execution.
- **Step 5:** But when the control comes to nSum(0), the condition (n == 0) becomes true and the statement **return 0** is executed.



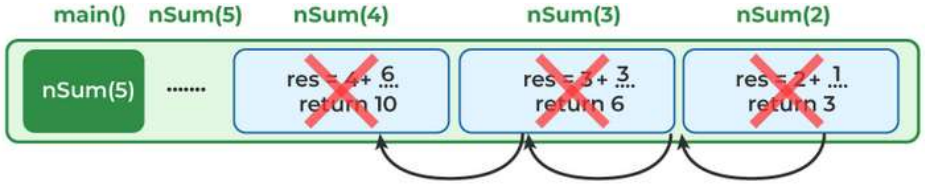
- **Step 6:** As the value is returned by the nSum(0), the stack frame for the nSum(0) will be destroyed, and using the instruction pointer, the program control will return to the nSum(1) function and the nSum(0) call will be replaced by value 0.



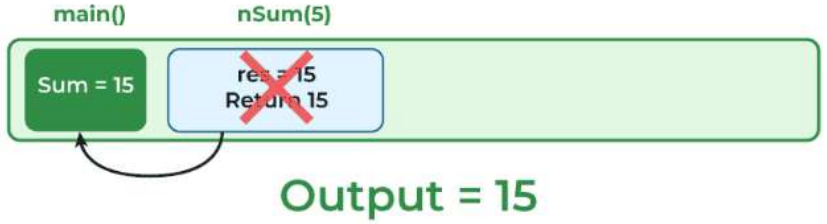
- **Step 7:** Now, in nSum(1), the expression **int res = 1 + 0** will be evaluated and the statement **return res** will be executed. The program control will move to the nSum(2) using its instruction pointer.



- **Step 8:** In nSum(2), nSum(1) call will be replaced by the value it returned, which is 1. So, after evaluating **int res = 2 + 1, 3** will be returned to nSum(3). Same thing will keep happening till the control comes to the nSum(5) again.



- **Step 9:** When the control reaches the nSum(5), the expression **int res = 5 + nSum(4)** will look like **int res = 5 + 10.** Finally, this value will be returned to the main() function and the execution of nSum() function will stop.



Drawbacks of recursion:

- has an overhead
- expensive in memory space and processor time
- each call involves processor overhead of call (changing program counter and saving return address)
- compared to this, iteration uses a repetition structure inside the function instead of calling it again and again
- still, recursive method at times gives logically more sound and more readable programs, hence it is used
- during expansion phase, each call requires another copy of function variables

Module-3

- Aggregate Data-types:
  - Arrays and Pointers
  - Structures
  - Dynamic data and Pointers
  - Dynamic arrays

Thanks