

# Fundamentals of Computer Programming

## Chapter 5 Modular Programming (Function in C++)



Chere L. (M.Tech)  
Lecturer, SWEG, AASTU<sup>1</sup>

- Introduction to modular programming
- Function declaration and definition
- Calling a function and Return types
- Function parameters (Value Vs. Reference)
- Parameter Passing
  - ✓ *by value*
  - ✓ *by reference*
- Default arguments
- Function Overloading
- Scope of variables (revision)
- Special functions (recursive functions, inline functions)
- Array in function (array as parameter, return array, call with array)

# 1. Introduction Modular Programming

## Modular programming

- Programming approach in a complex problem breaking down in to smaller manageable pieces
- The design of a program into individual components (modules) that can be programmed and tested independently.
- It is a requirement for effective development and maintenance of large programs and projects
- Procedures of a common functionality are grouped together into separate modules.
- A program therefore no longer consists of only one single part
- It is now divided into several smaller parts which interact and which form the whole program.

# Cont'd . . .

## Modular program

- A program consisting of interrelated segments (or modules) arranged in a logical and understandable form
- Modules in C++ can be classes or functions

## Why Modular programming?

- Easy to write a correct small function
- Code Re-usability – Write once and use multiple times
- Code optimization – No need to write lot of code
- Maintainability – Easily to debug, maintain/modify the program
- Understandability – Easy to read, write and debug a function

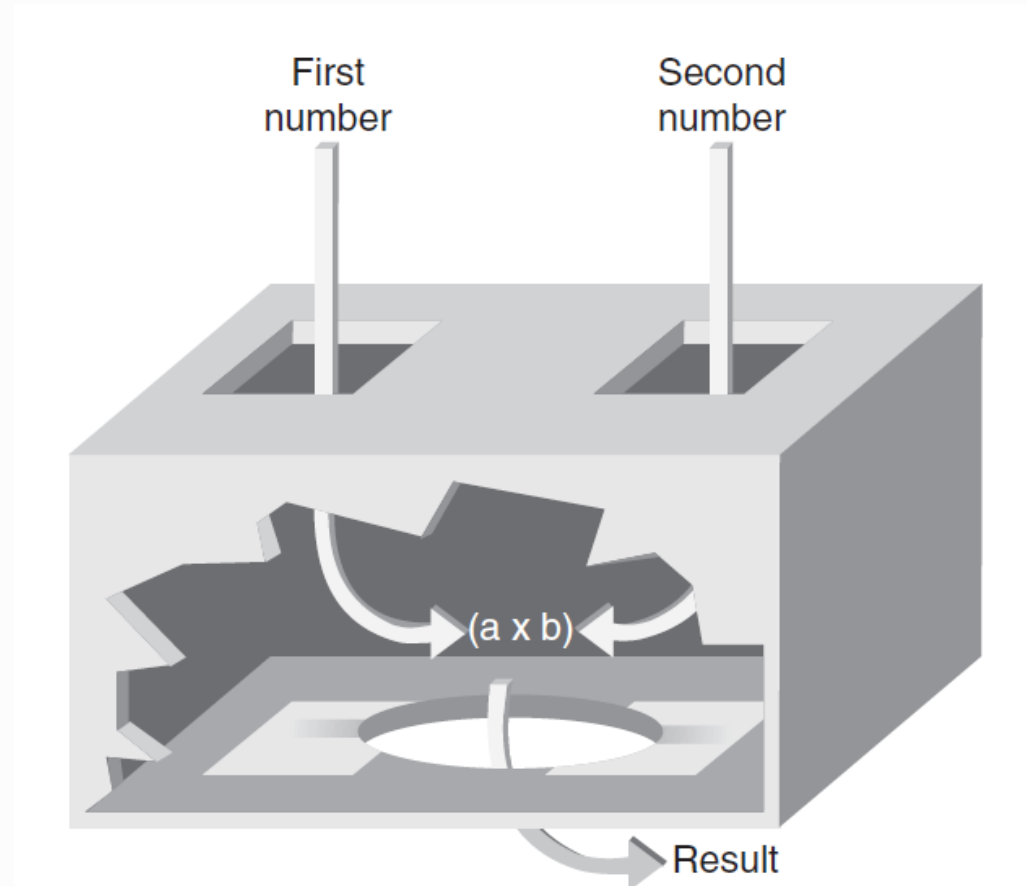
## 2. The concepts of Function

### Functions

- A function is a block of code (subprogram) that **performs a specific task**.
- Complicated tasks should be ***broken down into multiple functions***.
- Each can be called in turn when it needed
- Note:
  - Every C++ program has at least one function, main().

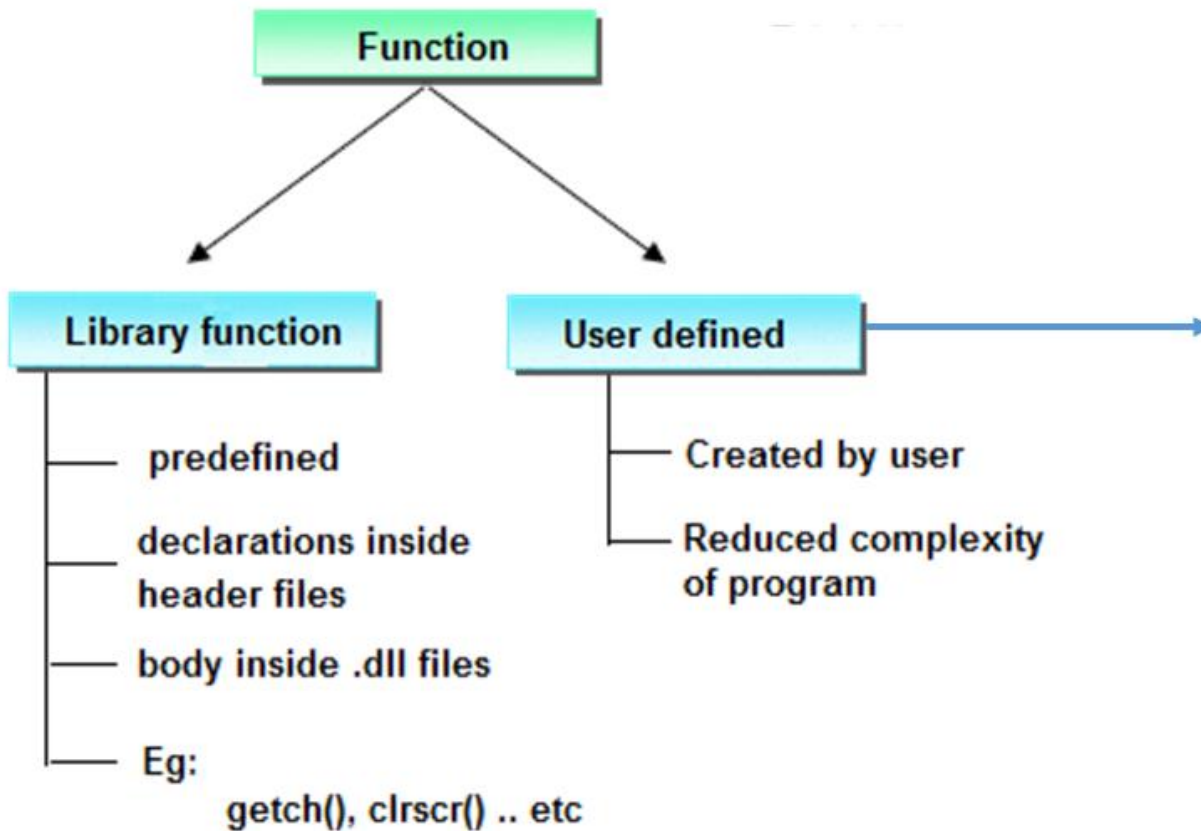
## Cont'd . . .

- A function can Accepts an input, act on data (process the input) and return a value (produces an output).
- A function's processing is encapsulated and hidden within the function



# Cont'd . . .

## Types of Functions



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

# 3. Function declaration and definition

## Components of a function

- Function name
- Function arguments (parameters)
- Function body
- Return type

## Creation of a function

### ▪ Function declaration

- The process of tells the compiler about a function name.
- Also called function prototype creation

### ▪ Function definition

- Give body of function (i.e. write logic inside function body).

# Cont'd . . .

There are three ways to declare a function:

- Write your prototype into a separate file, and then use the `#include directive` to include it in your program.
- Write the prototype inside your program before the `main()` function.
- Define the function before it is called by any other function.
  - ✓ The definition acts as its own declaration.

▪ **Declaration syntax:**

```
return_type  function_name(parameter);
```

▪ **Definition syntax:**

```
returnType functionName (parameter1, parameter2,...) {  
    // function body  
}
```

# Cont'd . . .

- A function may return a value.
- It refers to the data type of the value the function returns.
- It is optional (void).

- The name of function it is decided by programmer
- Should be meaningful valid identifier

*Return Type*      *Function Name*      *Parameters*

*Function Header* { `int add(int x, int y)`

*Function Body* { `int sum = x+y;`  
`return(sum);` }

- A value which is pass in function at the time of calling of function
- It is like a placeholder.
- It is optional.
- Parameter identifier is also optional

- The collection of statements

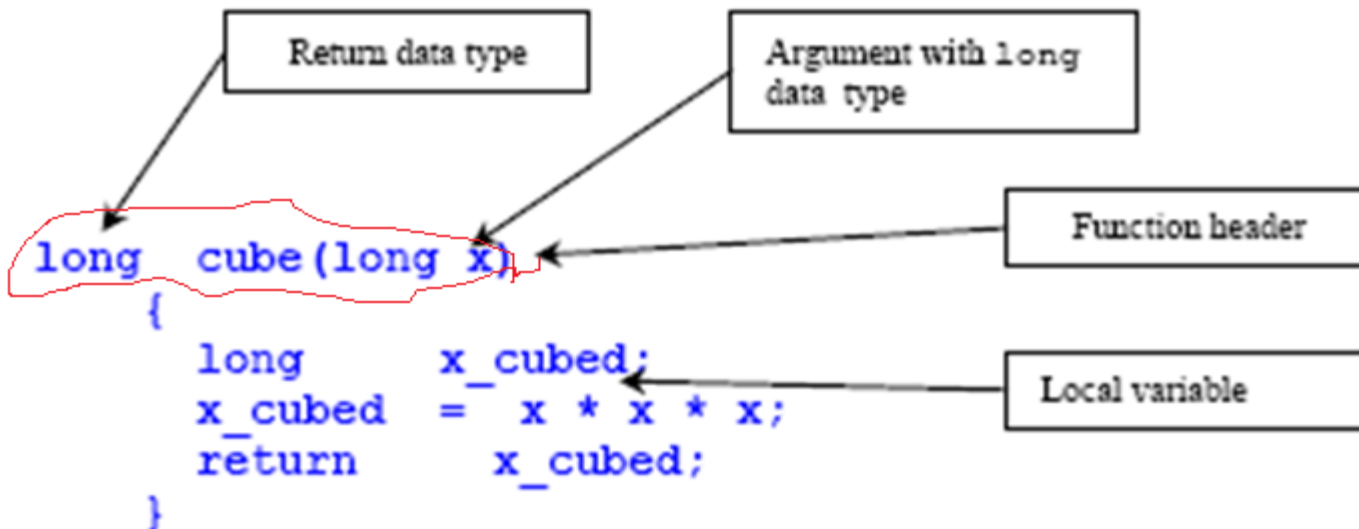
- Value returned by the function
- Single literal or expression

*return statement*

# Cont'd . . .

## Function Header

- First line of a function, which contains:
  - The type of data returned by the function (if any)
  - The name of the function
  - The type of data that must be passed into the function when it is invoked (if any)



# Cont'd . . .

## Examples

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

// display a number
void displayNum(int n1, float n2) {
    cout << "The int number is " << n1;
    cout << "The double number is " << n2;
}

int main() {

    int num1 = 5;
    double num2 = 5.5;

    // calling the function
    displayNum(num1, num2);

    return 0;
}
```

```
#include <iostream>

using namespace std;

// function prototype
int add(int, int);

int main() {
    int sum;

    // calling the function and storing
    // the returned value in sum
    sum = add(100, 78);

    cout << "100 + 78 = " << sum << endl;

    return 0;
}

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

# 4. Function calling, execution and return

## Function calling

- **Syntax :**  
`func_name(parameters );`  
or  
`Variable = func_name(parameters);`

## Function return

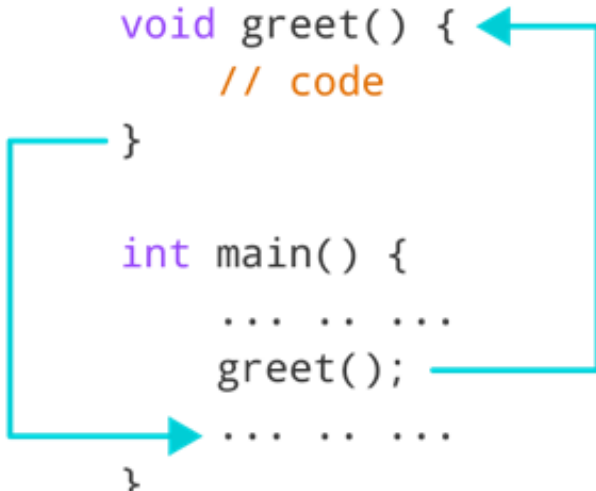
- **Syntax :**  
`return value/variable;`  
or `return expression;`

```
#include<iostream>

void greet() {
    // code
}

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

**function call**



```
#include<iostream>

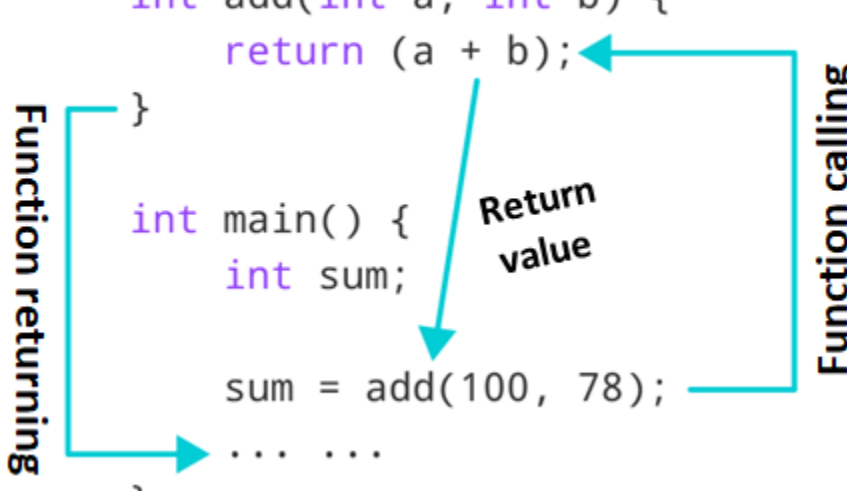
int add(int a, int b) {
    return (a + b);
}

int main() {
    int sum;
    sum = add(100, 78);
    ... ..
}
```

**Function returning**

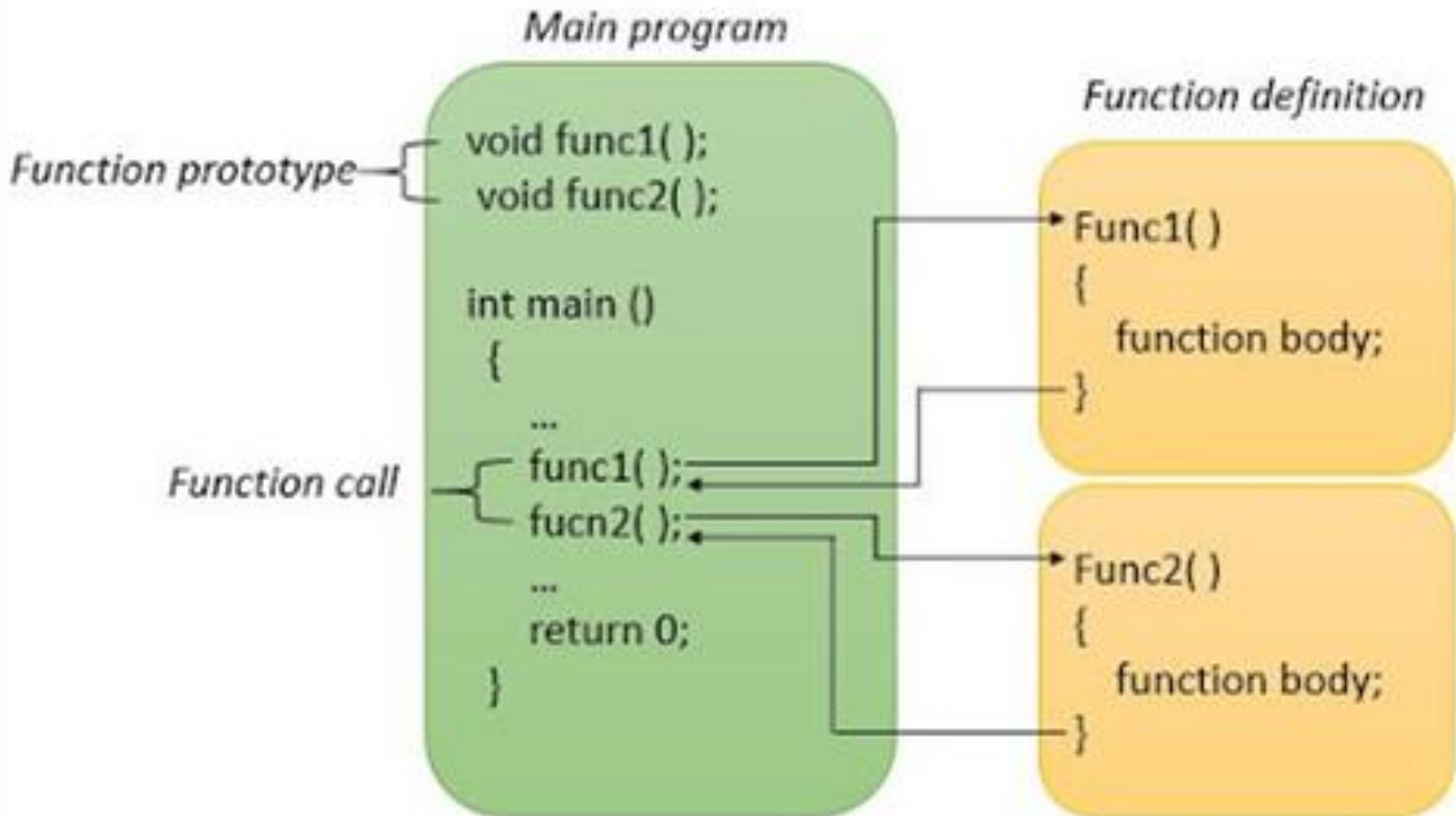
**Return value**

**Function calling**



# Cont'd . . .

## Function execution



# 5. Parameter passing

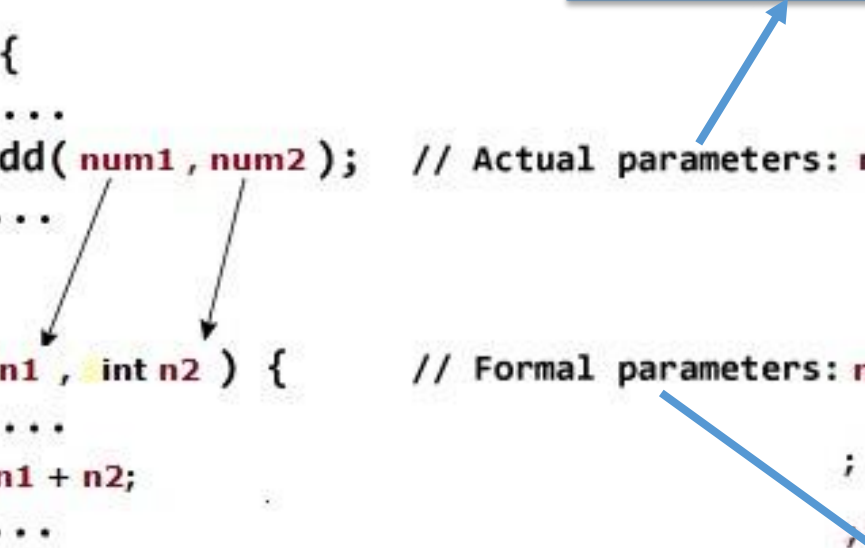
- Parameter is means by which functions are communicating and passing data
- Parameters are either Actual parameter or Formal Parameters

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

int add(int, int);

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

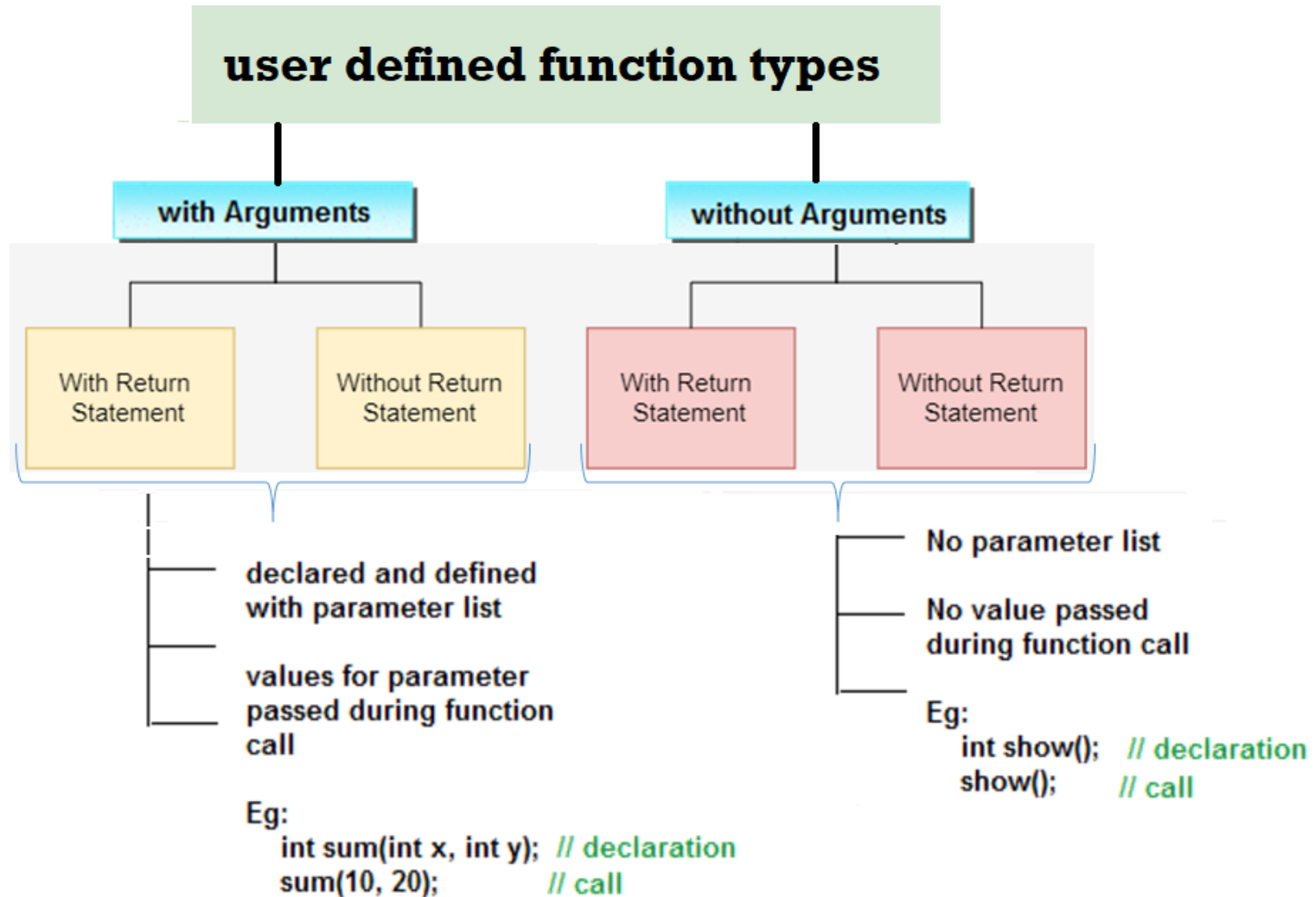
int add( int n1, int n2 ) { // Formal parameters: n1 and n2
    ... ..
    result = n1 + n2;
    ... ..
}
```



*The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environ*

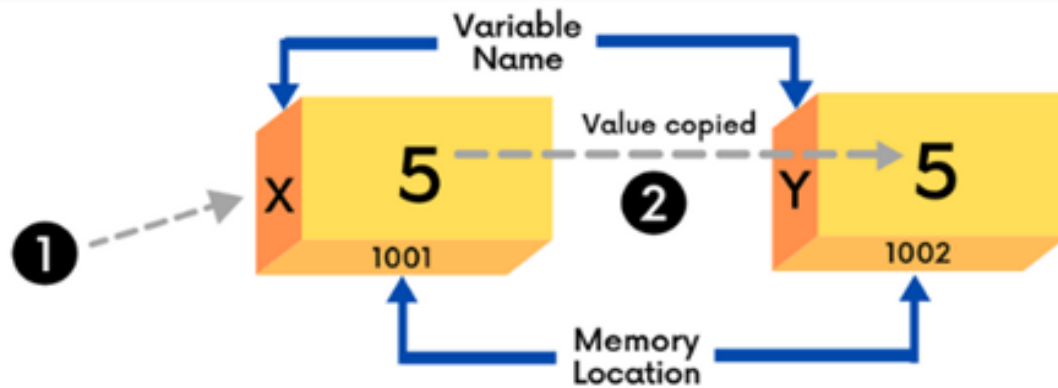
*A variable and its type as they appear in the prototype of the function or method.*

# Cont'd . . .

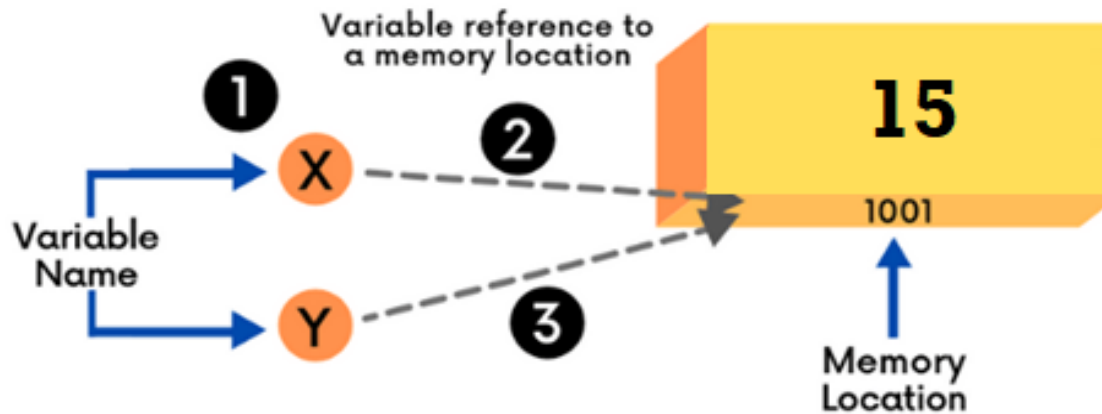


# Cont'd . . .

## Value Type Vs. Reference Type



**Value Type**



**Reference Type**

# Cont'd . . .

## Parameter passing by Value Vs. by Reference

*pass by reference*



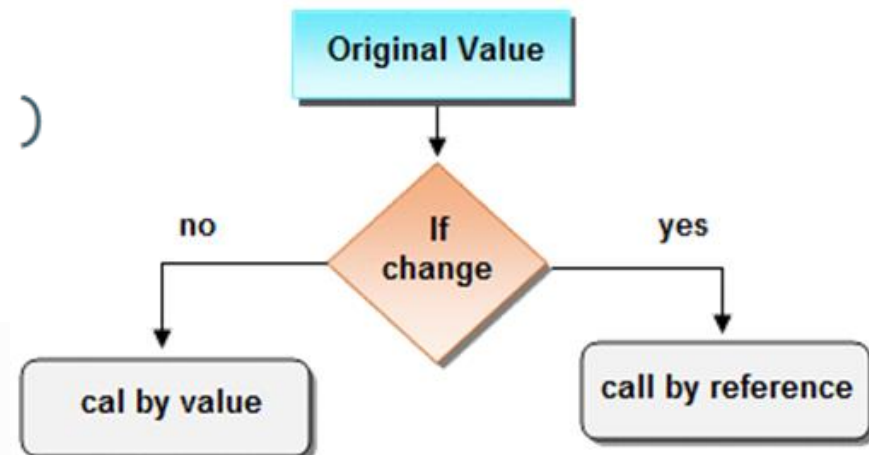
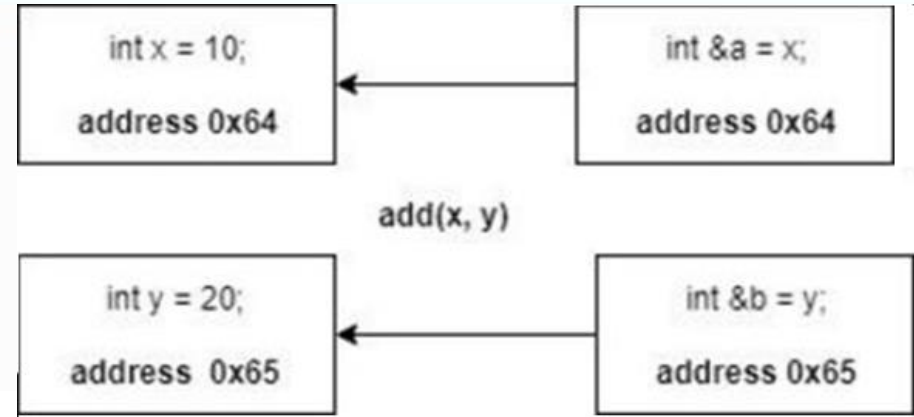
*fillCup( )*

*pass by value*



*fillCup( )*

www.penjee.com



# Cont'd . . .

## Parameter passing by Value Vs. by Reference

|   | call by value                                                                     | call by reference                                                                                         |
|---|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| 1 | This method copy original value into function as a arguments.                     | This method copy address of arguments into function as a arguments.                                       |
| 2 | Changes made to the parameter inside the function have no effect on the argument. | Changes made to the parameter affect the argument. Because address is used to access the actual argument. |
| 3 | Actual and formal arguments will be created in different memory location          | Actual and formal arguments will be created in same memory location                                       |

**Note:** By default, C++ uses call by value to pass arguments.

# Cont'd ...

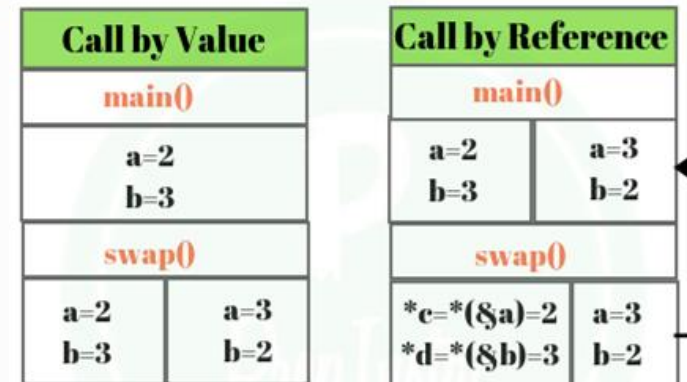
## Example 1: swapping two numbers

```
//C++ pass by value
void swap(int a, int b)
{ int temp;
  temp = a; a = b;
  b = temp;
}
```

```
int main() {
  int n = 10, m = 20;
  swap(n, m);
  /* no effect! */
}
```

```
// C++ pass by ref
void swap(int& a, int& b)
{ int temp;
  temp = a; a = b;
  b = temp;
}
```

```
int main() {
  int n = 10, m = 20;
  swap(n, m);
  /* n, m swapped! */
}
```



# Cont'd . . .

## "pass-by-value in C++" example:

```
#include <iostream>
using namespace std;
void swap(int first, int second); // prototype
int main()
{
    int x = 5;
    int y = 6;
    cout << "\nBefore: x = " << x << " y = " << y;
    swap(x, y);
    //were the integers swapped?
    cout << "\nAfter:  x = " << x << " y = " << y
        << endl;
    return 0;
}

void swap(int first, int second) {
    int temp;
    temp = first;
    first = second;
    second = temp;
}
```

Output:

Before: x = 5 y = 6

After: x = 5 y = 6

## "pass-by-reference in C++" example:

```
#include <iostream>
using namespace std;
void swap(int &first, int &second); // prototype
int main()
{
    int x = 5;
    int y = 6;
    cout << "\nBefore: x = " << x << " y = " << y;
    swap(x, y);
    //were the integers swapped?
    cout << "\nAfter:  x = " << x << " y = " << y
        << endl;
    return 0;
}

void swap(int &first, int &second) {
    int temp;
    temp = first;
    first = second;
    second = temp;
}
```

Output:

Before: x = 5 y = 6

After: x = 6 y = 5

# Cont'd . . .

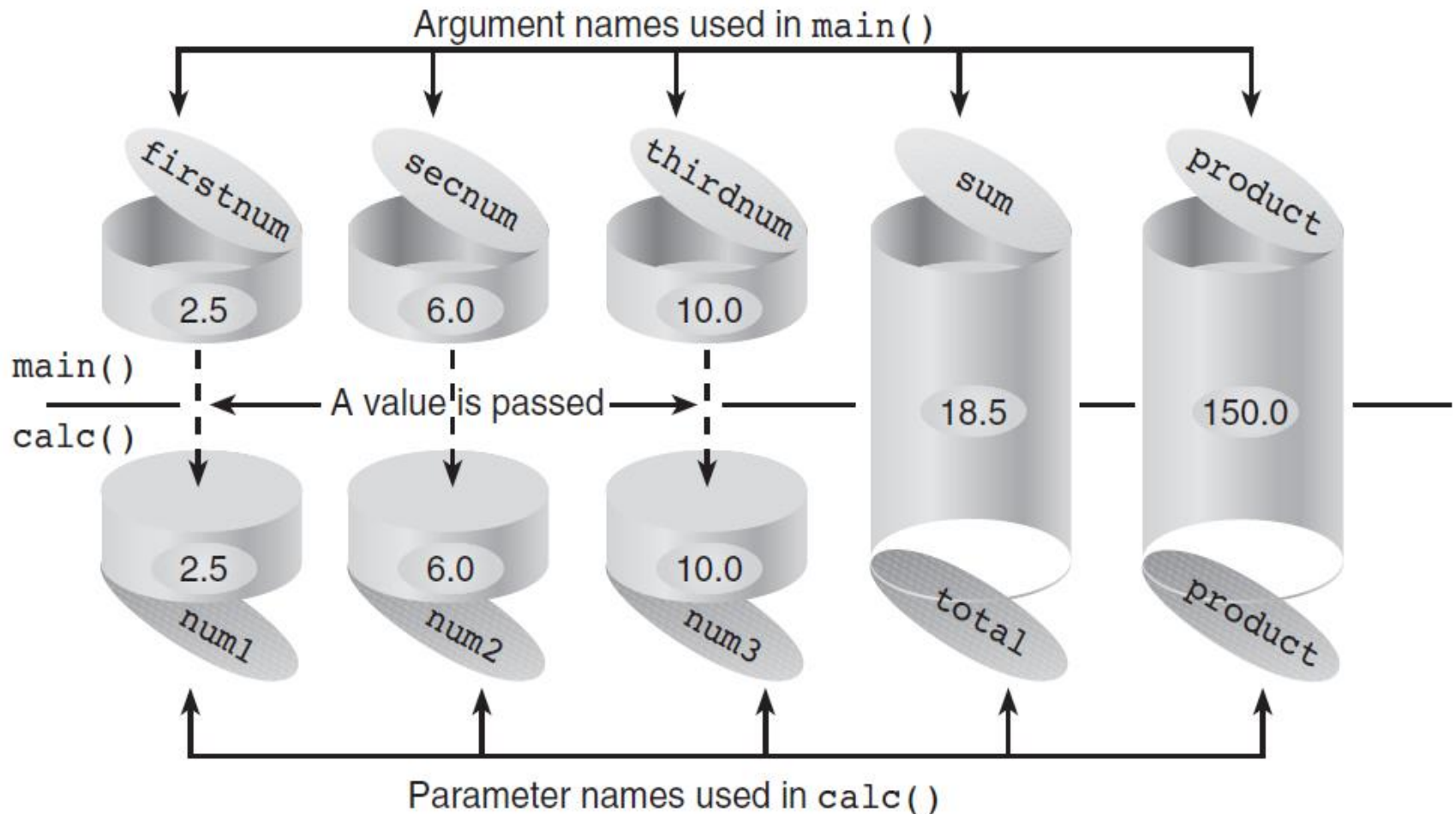
## Example 2

```
#include <iostream>
using namespace std;
void calc(double, double, double, double&, double&); // prototype
int main()
{
    double firstnum, secnum, thirdnum, sum, product;
    cout << "Enter three numbers: ";
    cin >> firstnum >> secnum >> thirdnum;
    calc(firstnum, secnum, thirdnum, sum, product); // function call
    cout << "\nThe sum of the numbers is: " << sum << endl;
    cout << "The product of the numbers is: " << product << endl;
    return 0;
}

void calc(double num1, double num2, double num3, double& total, double& product)
{
    total = num1 + num2 + num3;
    product = num1 * num2 * num3;
    return;
}
```

# Cont'd . . .

## Example 2 . . . .



# 6. 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**.

# Cont'd . . .

## ▪ Example 1

### Case 1 : No argument is passed

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

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

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

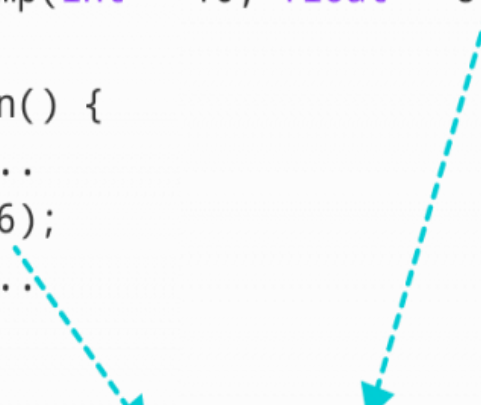


### Case 2 : First argument is passed

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

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

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



# Cont'd . . .

## ▪ Example 2

### 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
```

```
}
```



Error

# Cont'd . . .

## Things to Remember

- Once we provide a default value for a parameter, all **subsequent parameters must also have default values.**

- For example:

```
// Invalid
void add(int a, int b = 3, int c, int d);

// Invalid
void add(int a, int b = 3, int c, int d = 4);

// Valid
void add(int a, int c, int b = 3, int d = 4);
```

- If the *default arguments are provided in the function definition* instead of the function prototype, then the **function must be defined before the function call**

# 7. Function Overloading

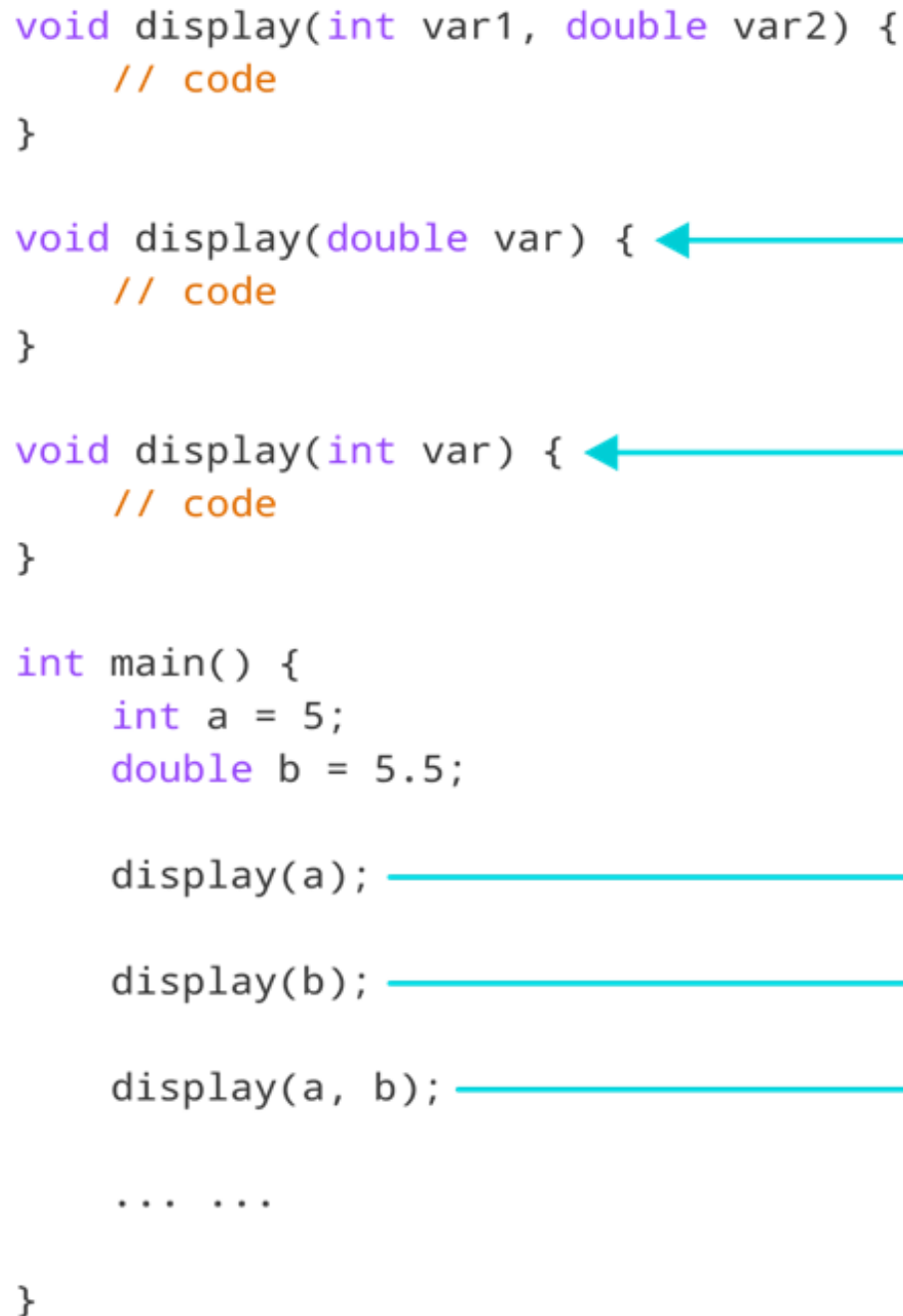
- In C++, two functions can have **the same name** if the *number and/or type of arguments passed is different*.
- These functions having the same name but different arguments are known as **overloaded functions**.
- For example:

```
void myFunction()  
void myFunction(int a)  
void myFunction(float a)  
void myFunction(int a, float b)  
float myFunction (float a, int b)
```

# Cont'd ...

## Example

```
void display(int var1, double var2) {  
    // code  
}  
  
void display(double var) {  
    // code  
}  
  
void display(int var) {  
    // code  
}  
  
int main() {  
    int a = 5;  
    double b = 5.5;  
  
    display(a);  
  
    display(b);  
  
    display(a, b);  
  
    ... ..  
}
```



# Cont'd . . .

## ▪ Example

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

int addNumbers(int, int);    // add ints
double addNumbers(double, double); // add doubles

int main(){
    cout <<"Sum of ints: "<<addNumbers(10, 20) << endl;
    cout <<"Sum of doubles: "<< addNumbers(10.2, 20.5) << endl;

    return 0;
}

int addNumbers(int a, int b){
    return a + b;
}

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

# 8. Revision on variable scope

- The scope of a variable is the portion of the program where the variable is valid or "known".

```
1 int foo;           // global variable
2
3 int some_function ()
4 {
5     int bar;       // local variable
6     bar = 0;
7 }
8
9 int other_function ()
10 {
11     foo = 1; // ok: foo is a global variable
12     bar = 2; // wrong: bar is not visible from this function
13 }
```

foo => I am a GLOBAL Variable.  
bar => I am a LOCAL Variable.

=> As You know in C++ scope of any variable is dependent on blocks or group of statement blocks ( Usually grouped by { } and use ; to separate multiple statements. )

=> GLOBAL Variable, can be used and looked for any-where in code.

=> LOCAL Variable exists only in their scopes until you make them GLOBAL , You can declare any LOCAL Variable as GLOBAL.

# Cont'd . . .

- What is the output of the following code fragment?

```
void thisFunc(int);
int m2;

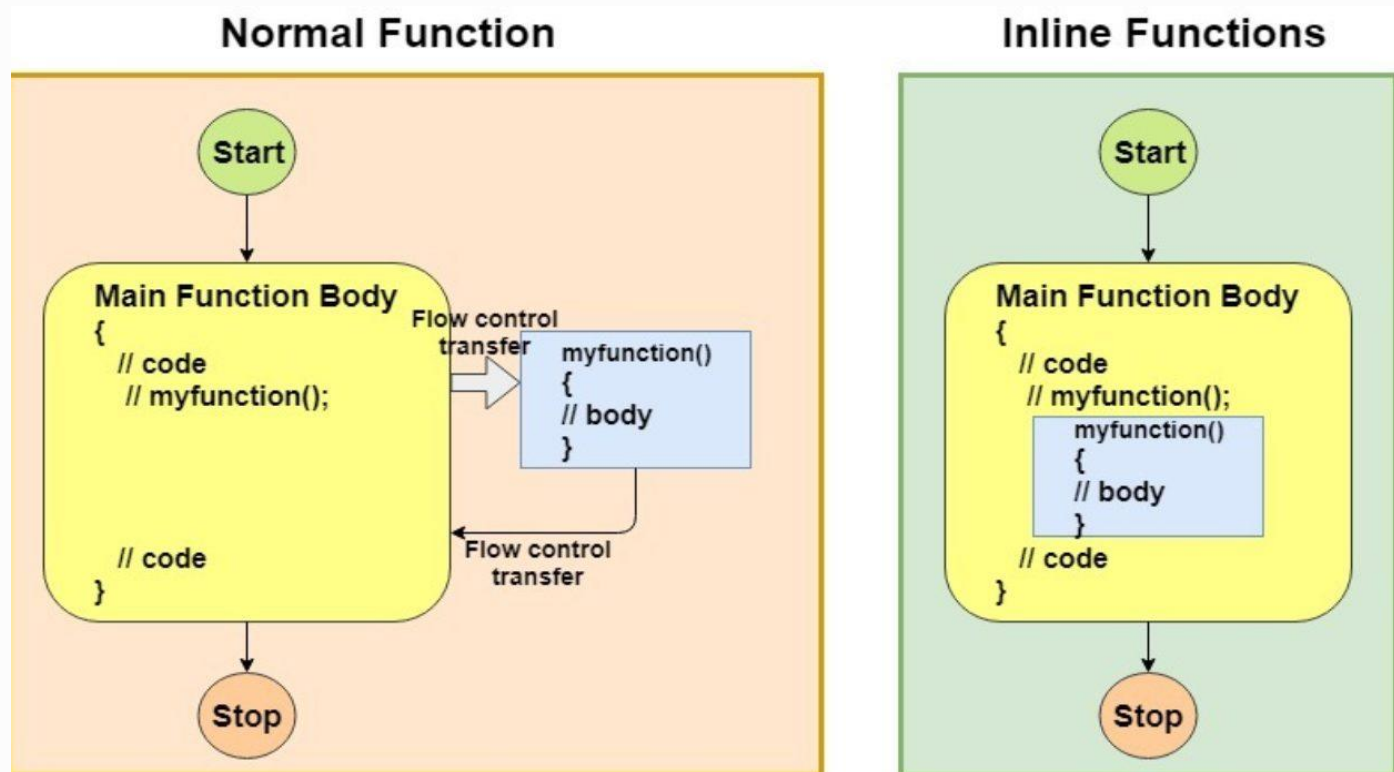
int main()
{
    int m1;
    m1 = 1;  m2 = 21;
    thisFunc(m1);
    cout << m1 << " " << m2 << endl;
    return 0;
}

void thisFunc (int a)
{
    int m2;
    m2 = a + 9;  a++;
    cout << a << " " << m2 << ", ";
}
```

Output:

# 9. Inline function

- If a function is **inline**, the compiler places a copy of the code of that function at each point where the function is called.
- To make any function inline function just preceded that function with **inline** keyword.



# Cont'd . . .

## Why use Inline function

- Whenever we call any function many time then, it take a lot of extra time in execution of series of instructions such as saving the register, pushing arguments, returning to calling function.
- For solve this problem in C++ introduce inline function.
- The main advantage of inline function is it make the program faster.

- **Example**

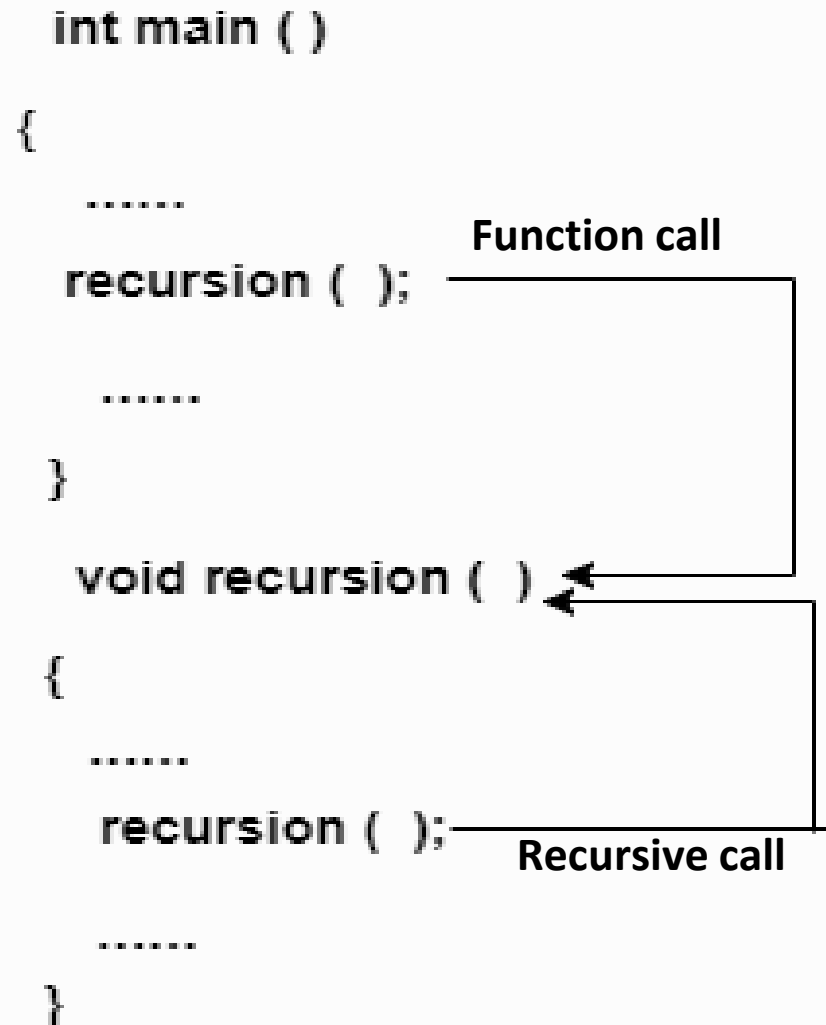
```
#include<iostream.h>
using namespace std;

inline void show()
{
    cout<<"Hello world";
}

int main()
{
    show();    //Call it like a normal function
    return ;
}
```

# 10. Recursive function

- A function that calls itself is known as a recursive function.
- The technique is known as recursion.



# Cont'd . . .

- Example:  
factorial finder  
function

```

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;
}

```

Diagram illustrating the recursive execution of the factorial function for  $n=3$ :

- Call 1:**  $n=3$ . The function calls  $\text{factorial}(2)$ . The return value is  $3 * 2 = 6$ , which is returned to  $\text{main}$ .
- Call 2:**  $n=2$ . The function calls  $\text{factorial}(1)$ . The return value is  $2 * 1 = 2$ , which is returned to the caller.
- Call 3:**  $n=1$ . The function returns 1, which is returned to the caller.

# 10. Function with Array

## (a) Calling function with array element

- Indexed variables can be arguments to functions
- Example: If a program contains these declaration

```
int i, n, a[10];
```

```
void my_function(int n);
```

- An array elements **a[0] through a[9]** are of type int, and calling the function as follow is legal:

```
my_function( a[ 0 ] );
```

```
my_function( a[ 3 ] );
```

```
my_function( a[ i ] );
```

### Note:

- In the same it works for 2D array and string

# Cont'd . . .

## (b) Array as formal parameter

- An entire array can be used as a formal parameter
- Such a parameter is called an **array parameter**
  - It is neither a call-by-value nor a call-by-reference parameter
  - However, behave much like call-by-reference parameters
- An array parameter is indicated using empty brackets or with array size in the parameter list

```
void fill_up(int a[ ], int size); or  
void fill_up(int a[5 ], int size);
```

### Note:

- Because a function *does not know the size of an array argument*, the programmer should include a formal parameter that specifies the size of the array as shown in the example above

# Cont'd . . .

## Example 1: passing 1D arrays to function

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

//function declaration:
void readArray(int arr[], int size);
void printArray(int arr[5], int size);
double getAverage(int arr[5]);

int main () {

    int balance[5];

    //calling function with array as an argument.
    readArray(balance, 5);
    printArray(balance, 5);
    double avg = getAverage(balance) ;

    // output the returned value, average
    cout<<"\n\nAverage value is: "<<avg<<endl;

    return 0;
}
```

Function declaration:  
To receive an array of int, arr[]  
as argument

```
C:\Users\Habesh\Documents\zz Exercises\array
Enter the 1 element: 57
Enter the 2 element: 94
Enter the 3 element: 61
Enter the 4 element: 37
Enter the 5 element: 85

The array elements are:
57 94 61 37 85

Average value is: 66.8

-----
Process exited after 9.555 seconds
Press any key to continue . . .
```

# Cont'd . . .

## Passing arrays to function .....

```
//function definition
void readArray(int arr[], int size){
    for (int i = 0; i < size; ++i)
    {
        cout<<"Enter the "<<i+1<<" element: ";
        cin>>arr[i];
    }
}

void printArray(int arr[5], int size){
    cout<<"\nThe array elements are: "<<endl;
    for (int i = 0; i < size; ++i)
    {
        cout<<arr[i]<<" ";
    }
}

double getAverage(int arr[5]){
    int sum = 0;
    for (int i = 0; i < 5; ++i) {
        sum += arr[i];
    }
    return (double(sum) / 5);
}
```

# Cont'd . . .

## (c) Returning an Array *(is it possible?)*

- Recall that functions can return a **value (single data element)** of type *int, double, char, . . .*
- Because array consist a set of the same type data elements, ***functions cannot return array.***
- However, an individual array element (single array element specified by index) can be returned.

- For example:

```
int myFunc(){  
    int myArray[10];  
    - - - - -  
    return myArray; //invalid  
}
```

//instead this is valid  
*return myArray[1];*

# Cont'd . . .

## Example 2: Passing 2D array to function

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

const int row=3, col=4;

void Read2Array(int arr[][col] );
void writing(int arr[row][col]);

int main ()
{
    int a[row][col];

    Read2Array(a);
    writing(a);

    return 0;
}
```

### Note:

- With 2D arrays, You can specify both dimensions
- However, the first dimension (number of rows) may be omitted, like you see on the **Read2Array()** function
- But the second dimension (number of columns) cannot be omitted.

# Cont'd . . .

2D array as  
parameter  
example ...

```
void Read2Array(int arr[][col] ) {
    int i,j;
    for (i=0; i<row; i++)
        for ( j=0; j<col; j++)
        {
            cout<<"Enter "<<i*j+1<<" Element: "
            cin >> arr[i][j];
        }
}

void writing(int arr[row][col]) {
    int i,j;
    cout<<"Array Elements are:\n"
    for (i=0; i<row; i++)
    {
        for ( j=0; j<col; j++)
            cout << arr[i][j] << '\t';
        cout << endl;
    }
}
```

# Cont'd . . .

## Example 3:

- Passing string to function as argument

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

void display(char s[]){
    cout<<"Entered char array is: "<<s<<endl;
}

void display(string s){
    cout<<"Entered string is: "<<s<<endl;
}

int main(){
    char str1[100];
    string str2;
    cout << "Enter a string: ";
    getline(cin, str1);
    cout << "Enter another string: ";
    cin.get(str, 100, '\n');

    display1(str1);
    display2(str2);
    return 0;
}
```

# Summary of Function

- Modular programming concept
- Function
  - what is function?
  - function components
  - using function
  - special functions
  - function with array

- Function name
- Function body
- Return type/value
- Function parameter

- ✓ What is parameter?
- ✓ Types of parameter
- ✓ Parameter passing
- ✓ Default parameter z

- ✓ Formal parameter
- ✓ Actual parameter

- Parameter passing
  - ✓ By value
  - ✓ By reference

- Inline function
- Recursive function
- Friend function
- Static function

- ✓ Array as parameter
- ✓ Calling function with array
- ✓ Return array

- Function declaration
- Function definition
- Function calling/returning
- Function execution

# Reading Resources/Materials

## *Chapter 9 & 10:*

- ✓ Diane Zak; An Introduction to Programming with C++ (8<sup>th</sup> Edition), 2016 Cengage Learning

## *Chapter 5:*

- ✓ Walter Savitch; Problem Solving With C++ [10th edition, University of California, San Diego, 2018

## *Chapter 6:*

- ✓ P. Deitel , H. Deitel; C++ how to program, 10th edition, Global Edition (2017)

---

Thank You  
For Your Attention!!

Any Questions

