Unit 2:
Some difference between C and C++:

Outline :

1. Functions
2. Function overloading
3. Macros and Inline Functions
4. Sharing variables between functions
    1. Global variables
    2. Pass by value
    3. Pass by address
    4. Pass by reference
5. new and delete operator
6. Function with default arguments

Topic 1:

FUNCTIONS

Topics to be covered related to functions

- What is a function?
- Types of functions:
  - Standard functions
  - User-defined functions
- function structure
  - Function signature
  - Function body
- Declaring and Implementing functions
- Sharing data among functions through function parameters
  - Value parameters
  - Reference parameters
- Scope of variables
  - Local Variables
  - Global variable

What is a function?

- 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 called as function
- A C++ function or a subprogram is simply a chunk of C++ code that has
  - A descriptive function name and parenthesis e.g.
    - *computeTaxes()* to compute the taxes for an employee
    - *isPrime()* to check whether or not a number is a prime number
  - A returning value
    - The c*omputeTaxes* function may return with a double number representing the amount of taxes
    - The *isPrime* function may return with a Boolean value (0 or 1)

Advantages of Functions

- A complex problem is often easier to solve by dividing it into several smaller parts (modules), each of which can be solved by itself for particular task. This is called ***procedure oriented*** programming.
- **`main()`** then uses these functions to solve the original problem.
- Functions separate the concept (what is done ?) from the implementation (how it is done ?).
- Functions make programs easier to understand debug.
- Functions can be called several times in the same program, allowing the code to be reused.

Standard Functions

- C++ language is equipped with a lot of functions which are known as standard functions (Predefined)
- These standard functions are groups in different libraries which can be included in the C++ program, e.g.
  - Math functions are declared in <math.h> library
  - Character-manipulation functions are declared in <ctype.h> library
  - C++ is shipped with more than 100 standard libraries, some of them are very popular such as <iostream.h> and <stdlib.h>, others are very specific to certain hardware platform, e.g. <limits.h> and <largeInt.h>

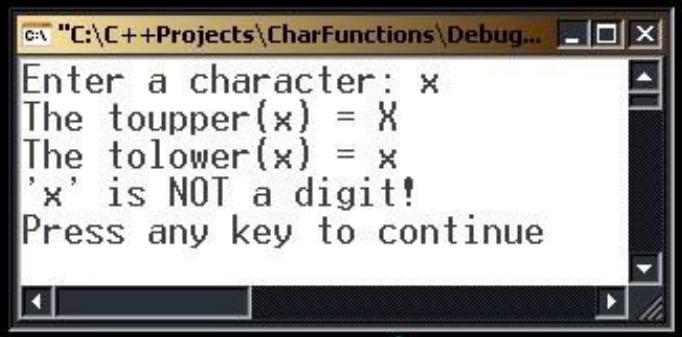
Example 1:
Using Standard Math Functions

```
#include <iostream.h>
#include <math.h>
void main()
{
 // Getting a double value
 double x;
 cout << "Please enter a real number: ";
 cin >> x;
 // Compute the ceiling and the floor of the real number
 cout << "The ceil(" << x << ") = " << ceil(x) << endl;
 cout << "The floor(" << x << ") = " << floor(x) << endl;
}
```

Example 2:
Using Standard Character Functions

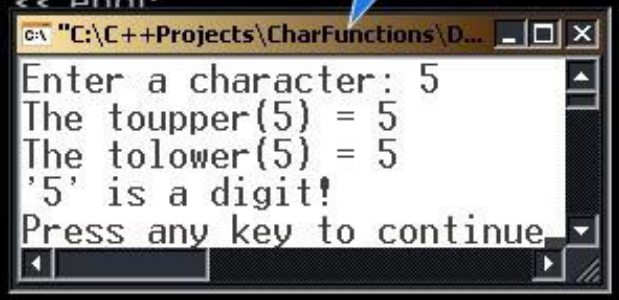
The function **isdigit**() is used to check that character is a numeric character or not. This function is declared in “ctype. h” header file.

```
#include <iostream.h>  // input/output handling
#include <ctype.h>   // character type functions
using namespace std;
void main()
{
 char ch;
 cout << "Enter a character: ";
 cin >> ch;
 cout << "The toupper(" << ch << ") = " << (char) toupper(ch) << endl;
 cout << "The tolower(" << ch << ") = " << (char) tolower(ch) << endl;
 if (isdigit(ch))
        cout << "'" << ch <<"' is a digit!\n";
 else
        cout << "'" << ch <<"' is NOT a digit!\n";
}
```



Explicit type casting is required as toupper function return int value

Explicit casting



User-Defined Functions

- Although C++ is shipped with a lot of standard functions, these functions are not enough for all users, therefore, C++ provides its users with a way to **define their own functions** (or **user-defined function**)

For example:

<math.h> library does not include a standard function that allows users to round a real number to the i^{th} digits, therefore, we must declare and implement this function ourselves

How to define a function?

- C++ function can be define in two steps (preferably but not mandatory)
  - Step #1 – declare the *function signature* (function prototype) before the main function of the program
  - Step #2 – define the function after the main function

Or

  - Define the function body before the main function, in this case function prototype is not mandatory.

Structure of a function

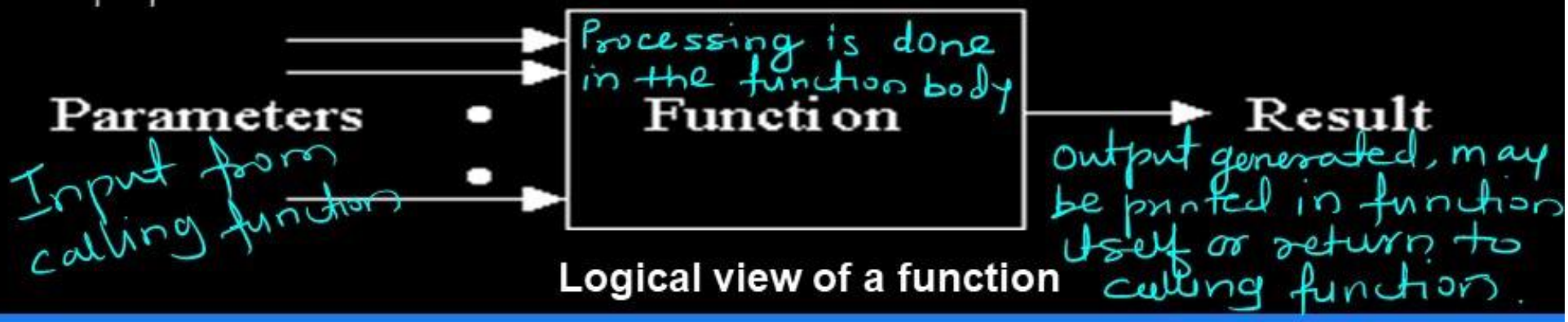
- A C++ function consists of two parts
  - The function header, and
  - The function body

The function header has the following syntax:

<return value> <name of the function> (<parameter list>)

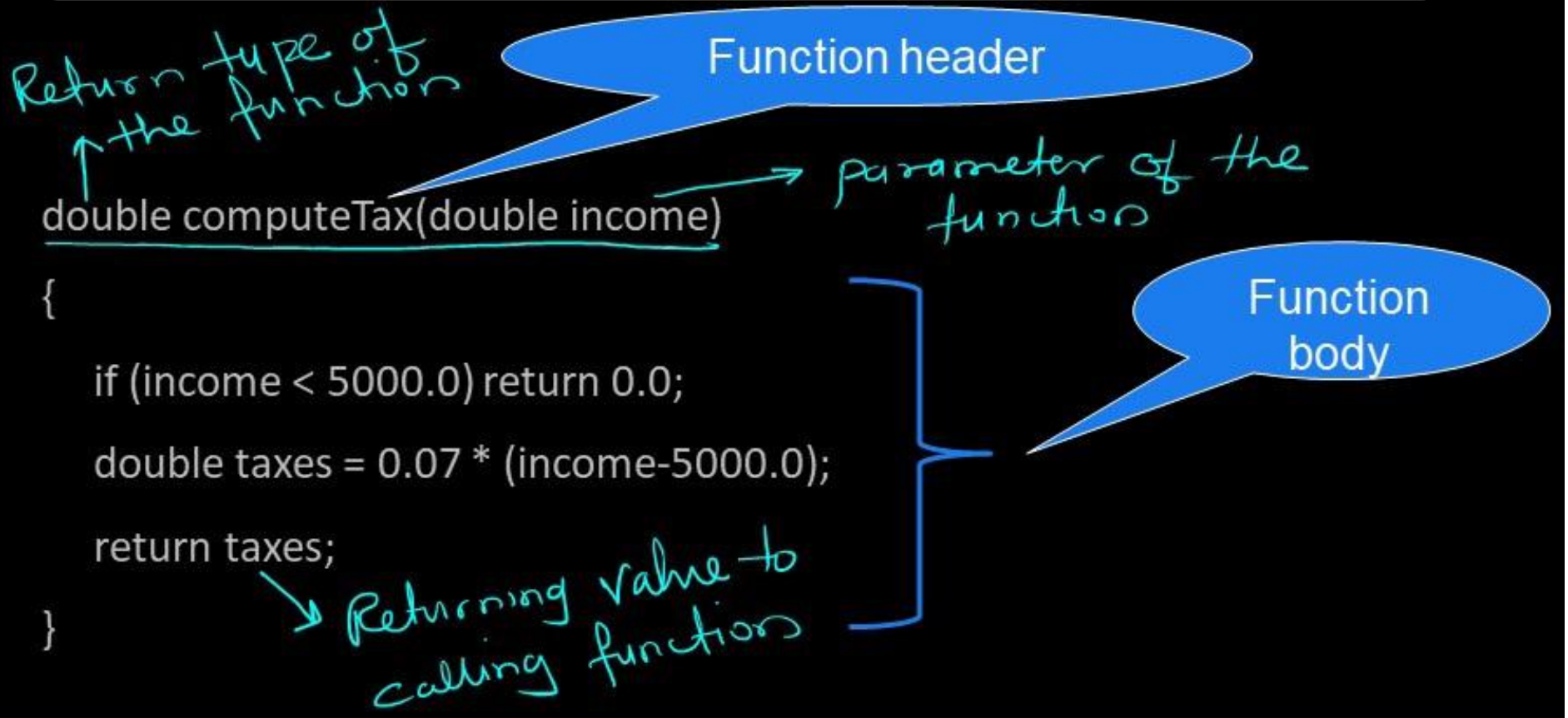
The function body is simply a C++ code enclosed between

{ }



Logical view of a function

Example 1: using User-defined function



Types of function definition

Type 1:

- *Non value-returning and No parameters to functions* : simple function

```
void function-name()
{
 constant declarations
 variable declarations

 other C++ statements
}
```

Type 2:

- *Non value-returning with parameter to functions* :

```
void function-name(parameter list)
{
 constant declarations
 variable declarations

 other C++ statements
}
```

Types of function definition

Type 3:

- *value-returning with no parameters functions*

```
int function-name()
{
  constant declarations
  variable declarations

  other C++ statements
 return value;
}
```

Return type

NO arguments

Returning value to calling function

Type 4:

- *value-returning with parameters functions* : The argument names in the function header are referred to as *formal parameters*

```
int FindMax (int x, int y)
{
 int maximum;

 if(x>=y)
   maximum = x;
 else
   maximum = y;

 return maximum;
}
```

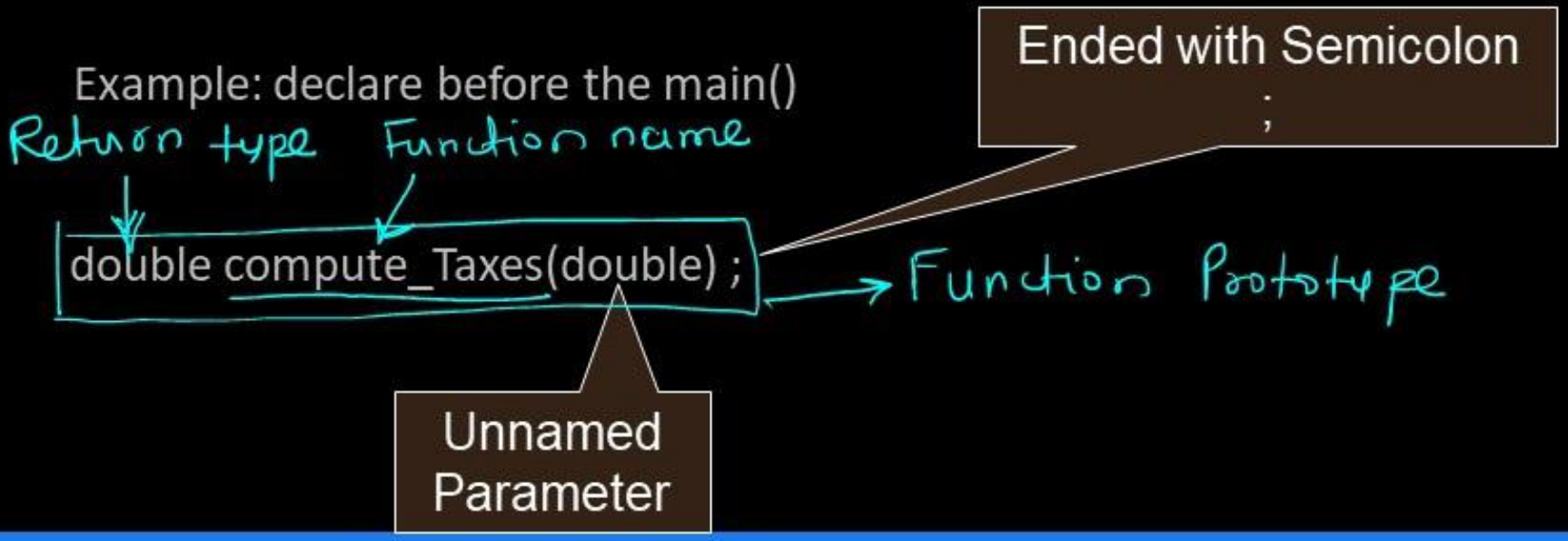
Return type

With argument

Function prototype (Function signature)

- The function prototype is actually similar to the function header except in two aspects:
  - The parameters names may not be specified in the function signature. However types of parameters are necessary.
  - The function prototype must be ended by a semicolon

Example: declare before the main()



Why function prototype is required?

- The use of function prototypes permits *error checking* of data types by the compiler.
- It also ensures conversion of all arguments passed to the function to the declared argument data type when the function is called.
- For Information Hiding
  - If you want to create your own library and share it with your customers without letting them know the implementation details, you should declare all the function signatures in a header (.h) file and distribute the binary code of the implementation file
- For Function Abstraction
  - By only sharing the function signatures, we have the liberty to change the implementation details from time to time to
    - Improve function performance
    - make the customers focus on the purpose of the function, not its implementation

Example 1: function prototype

```
#include <iostream>
#include <string>
using namespace std;
  double getIncome(char *prompt);
  double compute_Taxes(double);
  void printTaxes(double);
void main()
{
  // Get the income;
  double income = getIncome ("Please enter the
  employee income: ");
  // Compute Taxes
  double taxes = compute_Taxes(income);
  // Print employee taxes
  printTaxes(taxes);
}
```

Function Prototype declaration

```
double compute_Taxes (double income)
{
  if (income<5000) return 0.0;
  return 0.07*(income-5000.0);
}
double getIncome(char *prompt)
{
  cout << prompt;
  double income;
  cin >> income;
     return income;
}
void printTaxes (double taxes)
{
  cout << "The taxes is $" << taxes << endl;
}
```

Function Definition after main

Calling a function : Syntax

- A function is *called* by specifying its name followed by its arguments.
- Calling of a function has to be ended with semicolon

Following two type are used to call the function:

1. **Non-value returning functions:**
   - *function-name (data passed to function);*
   - `Example : minimum(a,b);`
2. **Value returning functions:**
   - *results = function-name (data passed to function);*
   - `Example: x = minimum(a,b);`

Example 1: Calling a function

```
#include <iostream>
int FindMax(int, int);          // function prototype

int main()
{
 int firstnum, secnum, max;
 cout << "\nEnter two numbers: ";
 cin >> firstnum >> secnum;
 max = FindMax (firstnum, secnum);        // the function is called here
 cout << "The maximum is " << max << endl;
 return 0;
}
```

Return Value stored in max

The argument names in the function call are referred to as *actual parameters*

Topic 2:

FUNCTIONS OVERLOADING

Function Overloading

- Function overloading is a **C++ programming** feature that allows programmer to have more than one function having same name used for different task of similar nature with different parameter list.
  - Different parameter list means: The Data type, Number and Sequence of the parameters.
- The compiler must be able to determine which function to use based on the number and data types 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 also called as **compile-time polymorphism**.
  Lets see the rules of overloading: we can have following functions in the same scope.

Rules of overloading

- Lets see the rules of overloading: we can have following functions in the same scope.

```
int sum(int num1, int num2)

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

int 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.
  - Overloaded functions may or may not have different return types
- Example: These functions would have different parameters types and number is same

```
int sum(int num1, int num2) { }

double sum(double num1, double num2) { }
```

Example1 : Function Overloading

Program to compute sum of values with different numbers of parameters

```
include <iostream>
using namespace std;
int sum(int num1,int num2)
{       return num1+num2;    }
int sum(int num1,int num2, int num3)
{    return num1+num2+num3;    }
int main(void) {
cout<<sum(20, 15)<<endl;
cout<<sum(81, 100, 10);
return 0; }
```

→ First Function with Two parameters

→ second function with Three parameters

main function

Calling of a function

Example 2: Function Overloading

Program to compute absolute value // Works with different types of parameters for both int and float

```
#include <iostream>

using namespace std;

float absolute(float var)        // function with float type parameter
{ if (var < 0.0) var = -var; return var; }

int absolute(int var)            // function with int type parameter
{ if (var < 0) var = -var; return var; }

int main() {                     // call function with int type parameter
cout << "Absolute value of -5 = " << absolute(-5) << endl; // call function
                                                              with float type parameter
cout << "Absolute value of 5.5 = " << absolute(5.5f) << endl;
return 0; }
```

Function one

Function two

Different types of parameters

Example 3: Function Overloading

```
// Program to compute area of circle, rectangle and triangle.
#include <iostream>
float area(float r)              // function with float type parameter
{ float A;  A = 3.14 * r * r;  return A; }
int area(int l, int b)
{ int A;  A = l * b;  return A;}
float area( int b, float h)
{ float A; A = (b*h) /2;   return A; }
int main() {          // call function with int type parameter
cout << "Area of circle= " << area(3.4) << endl;
cout << "Area of rectangle= " << area(5 , 4) << endl;
cout << "Area of triangle = " << area(5 , 6.2) << endl;
}
```

Advantage 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.

Topic 3:

MACROS AND INLINE FUNCTIONS

Macros

Pre processor Directive

1. During compilation pi will get replaced by 3.14 So expression will be → A = 3.14 * r * r;
2. No memory access will be done for pi, so saves execution time
3. Pi will get replace in both the functions by 3.14

```
#include <iostream>

# define pi 3.14

void areaCircle (float r)
{ float A;  A = pi * r * r;
  cout<< A; }
void volumeSphere (float r)
{ float V = 4.0 /3.0 * pi *r *r * r ;
  cout<< V; }

int main()
{
areaCircle(3.4);
volumeSphere (5.2);
}
```

define value of pi as 3.14

replaced with 3.14

- Macros are pre-processor directives – it directs compilers to perform certain task
- They are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code.
- The '#define' directive is used to define a macro.
- Let us now understand the macro definition with the help of a program:

Macro functions

- **#define** can be used to make **macro functions** that will be substituted in the source before compilation.
- A pre-processor function declaration comprises a macro name immediately followed by parentheses containing the function's argument.
- Do not leave any space between the name and the parentheses.
- The declaration is then followed by the function definition within another set of parentheses.
- For example, a pre-processor macro function to give largest of value of the two looks like this:

```
#define MAX(a,b) (a > b ? a : b)
```

Macros functions with arguments

```
#include <stdio.h>

#define AREA(l, b) (l * b) // macro with parameter

int main()
{
    int l1 = 10, l2 = 5, area;

    area = AREA(l1, l2);

    cout<<"Area of rectangle is:" <<area;

    return 0;
}
```

Expression of macro AREA

parameters to macro AREA

Returning result to area variable

- Arguments can be pass to macros. Macros defined with arguments works similarly as functions. They are also called as macro functions
- When use macros like function, it always return the value as result
- Let us understand this with a program:

Limitations of macro function

- When we use macro functions, however, unlike regular functions, they do not perform any kind of type checking.
- Macro functions do not allow complex expressions within the body.
- Because of this drawbacks, **inline functions** are usually preferable to macro functions.
- But because macros directly substitute their code, they reduce the overhead of a function call.

Why inline function?

- Let's first understand why inline functions are used and what is the purpose of inline function?
- When the program executes the function call instruction, the CPU perform following task
  1. Stores the memory address of the instruction following the function call
  2. Copies the arguments of the function on the stack and finally transfers control to the specified function.
  3. The CPU then executes the function code.
  4. Stores the function return value in a predefined memory location/register and returns control to the calling function.
- This can become overhead if the execution time of function is less than the switching time from the caller function to called function.

Why inline function?

- For functions that are large and / or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. → in such cases inline function is of no use
- However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code.
- This overhead occurs for small functions because execution time of small function is less than the switching time.
- C++ provides an inline functions to reduce the function call overhead. and save execution time

What is inline function?

```
#include<iostream.h>

inline void add (int a, int b)
{ int c = a+ b;   cout<<c; }

int main()
{
int x, y, z;
cin>>x>>y;
add(x , y);
}
```

Function definition

keyword is used

During compilation code of the function is substituted in main function

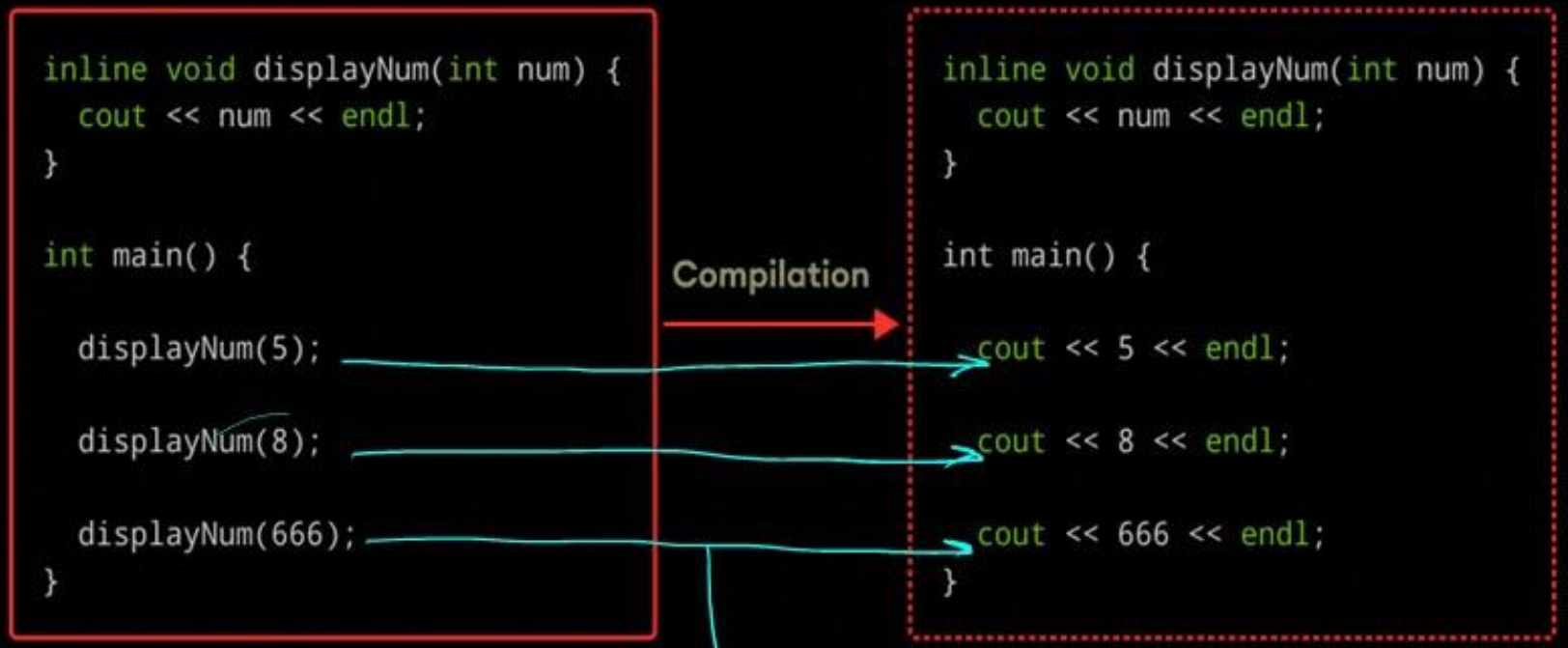
substituted code could be:

```
c = a+b
cout<<c
```

No function call done at the time of execution

- Inline function is a function that is expanded in line when it is called in the calling function.
- When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call.
- This substitution is performed by the C++ compiler at compile time.
- Inline function may increase efficiency if it is small in terms of reducing execution time of the program

Example of code after compilation



Code before compilation

→ displayNum function called three times in main.

Code after compilation

→ Every function call has been replaced by cout;

Example: 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)<<end;
return 0;
}
```

- We use the keyword *inline* to define user-defined functions
  - Inline functions are very small functions, generally, one or two lines of code
  - Inline functions are very fast functions compared to the functions declared without the inline keyword

Summary of inline function

- **Characteristics:** It is just a request to compiler to replace the code in calling function.

  Ignore the request if I am using:

  1. Loops in the code
  2. Decision making statements in the code like "switch case"
  3. goto statement.
  4. Function is recursive
  5. Function has static variable.

- **Advantages :**

  1. Function call overhead is reduced
  2. Reduce push/pop operation in the stack
  3. No return call transfer
  4. we can use the inline function over macros

Topic 4:

SHARING (DATA) VARIABLES BETWEEN FUNCTIONS

Sharing variables between functions:

- There are two ways to share data among different functions
  I. Using global variables (very bad practice, due to problem of data inconsistency)
  II. Passing data through function parameters
    1. Value parameters
    2. Reference parameters

C++ Variables

- A variable is a place in memory that has
  - A name or identifier (e.g. income, taxes, etc.)
  - A data type (e.g. int, double, char, etc.)
  - A size (number of bytes)
  - A scope (the part of the program code that can use it)
    - Global variables – all functions can see it and using it
    - Local variables – only the function that declare local variables see and use these variables
  - A life time (the duration of its existence)
    - Global variables can live as long as the program is executed
    - Local variables are lived only when the functions that define these variables are executed

I. Using Global Variables

```
#include <iostream>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
 f2();
 cout << x << endl;
}
```

Global Variable :- It can be accessed in all the functions which are defined below this statement.

:- It can be accessed in f1(), f2() and main()

I. Using Global Variables

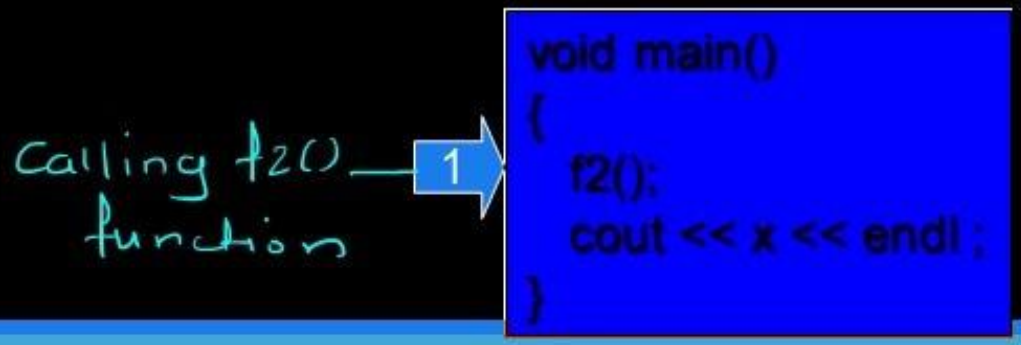
x 0

```
#include <iostream.h>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```

I. Using Global Variables

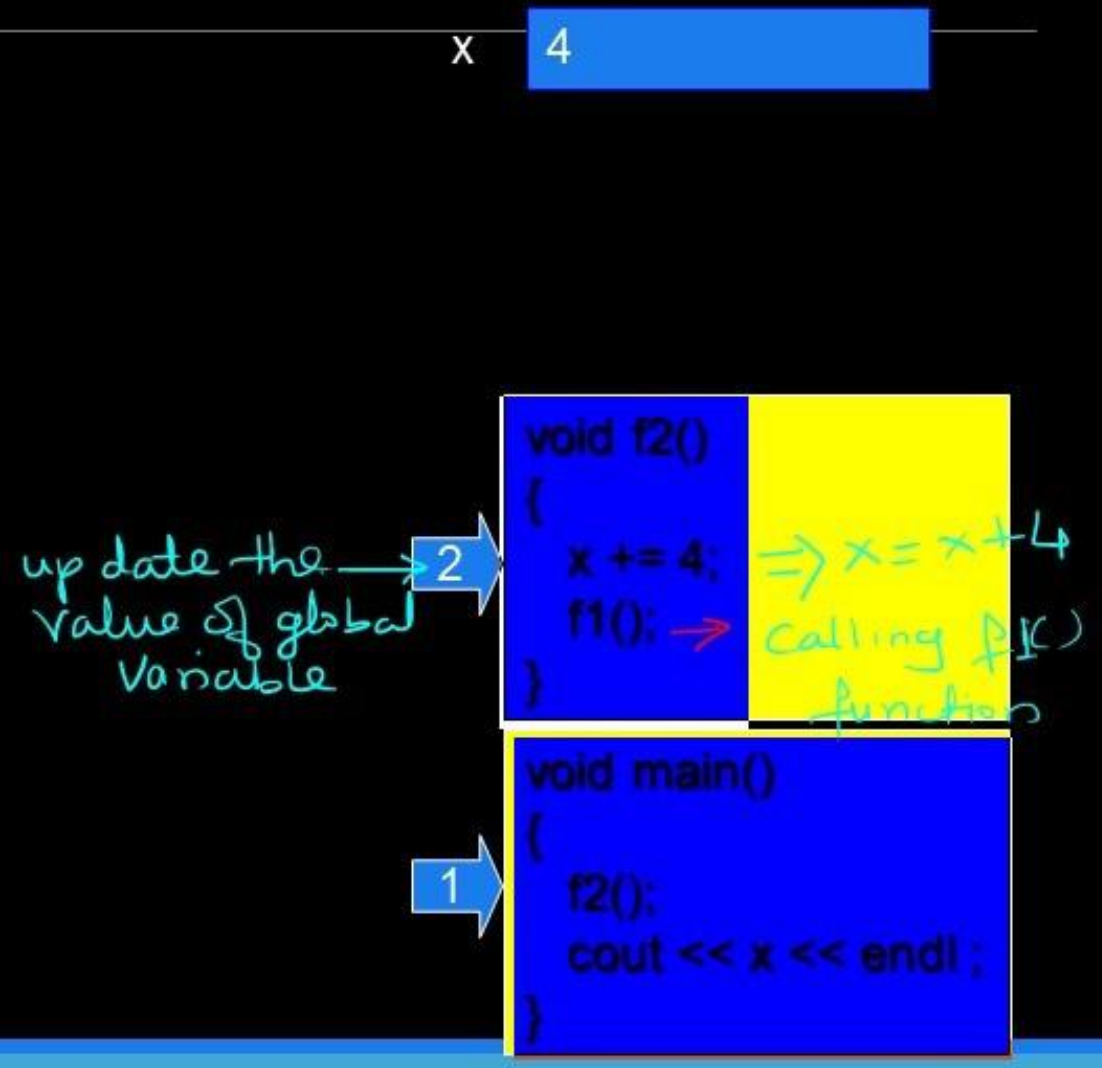
x 0

```
#include <iostream>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```



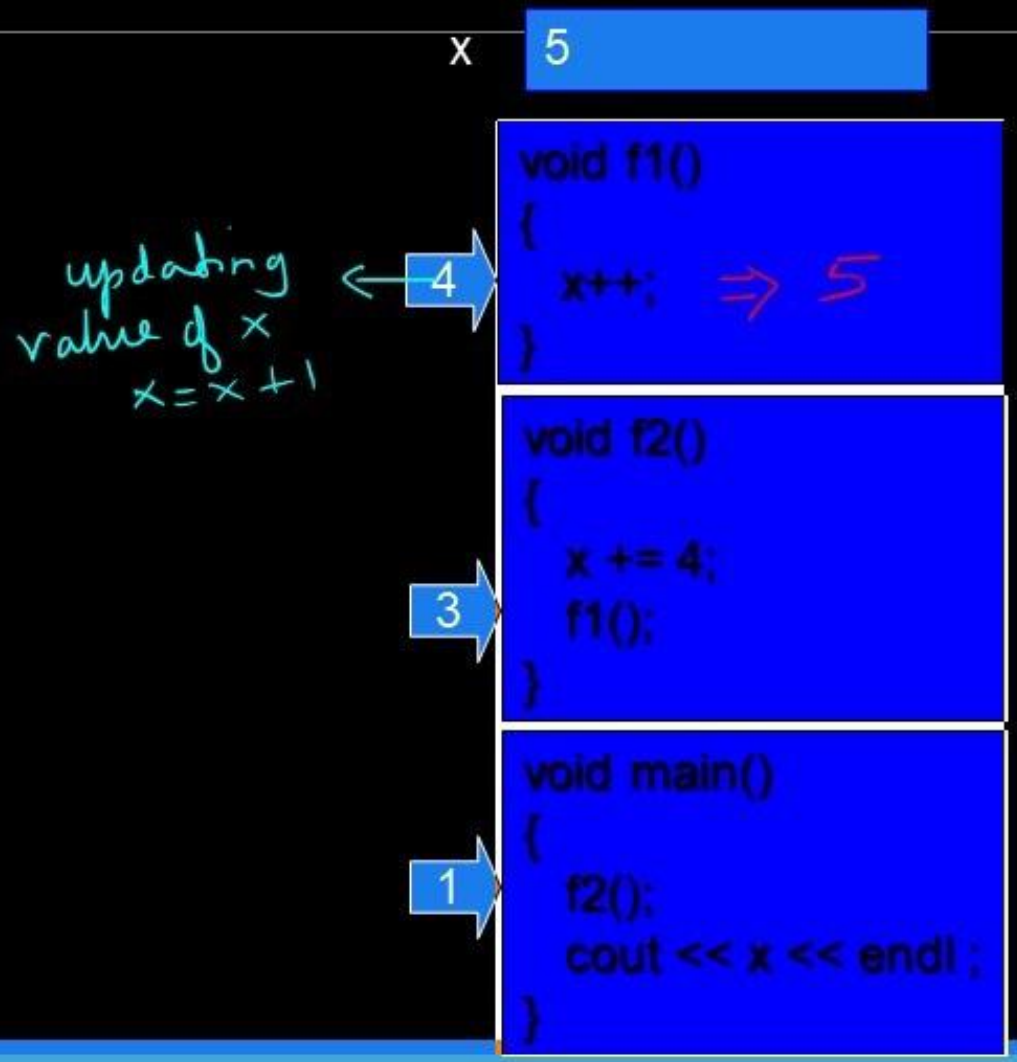
I. Using Global Variables

```
#include <iostream>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```



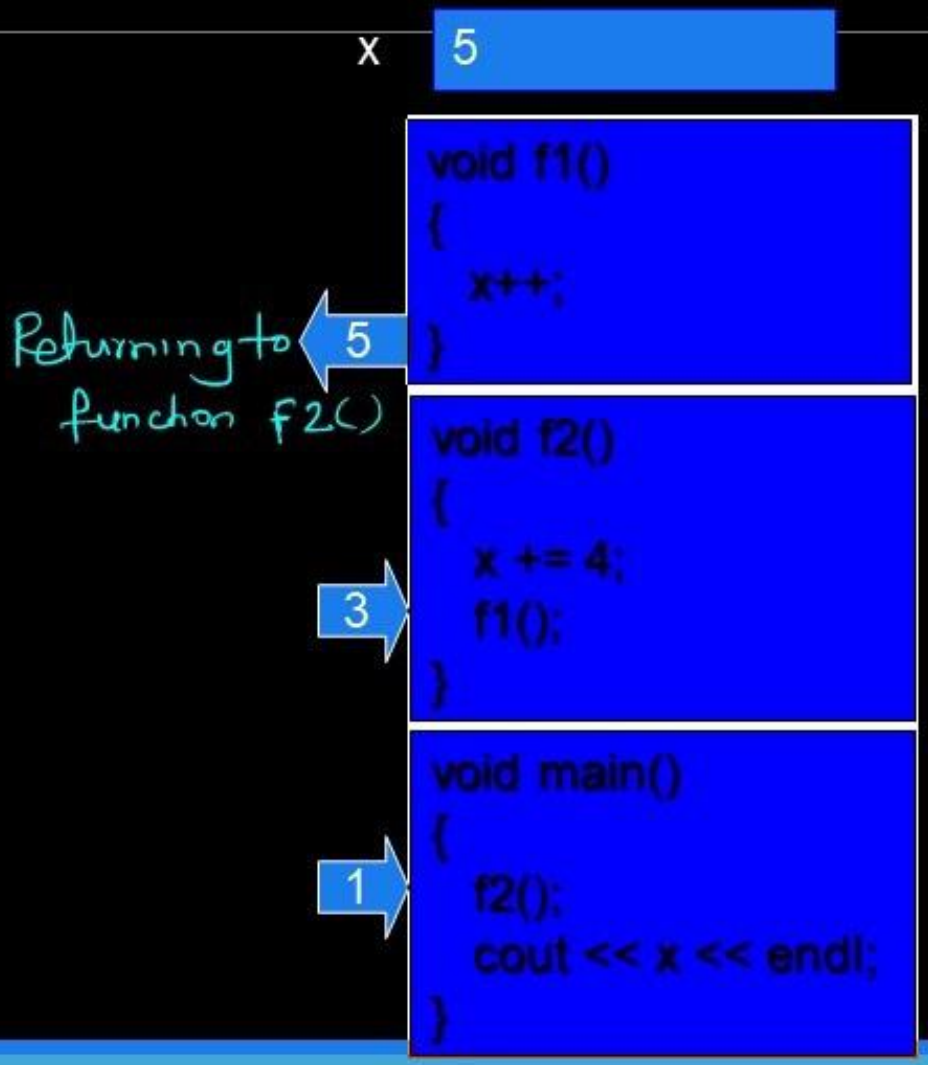
I. Using Global Variables

```
#include <iostream>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```



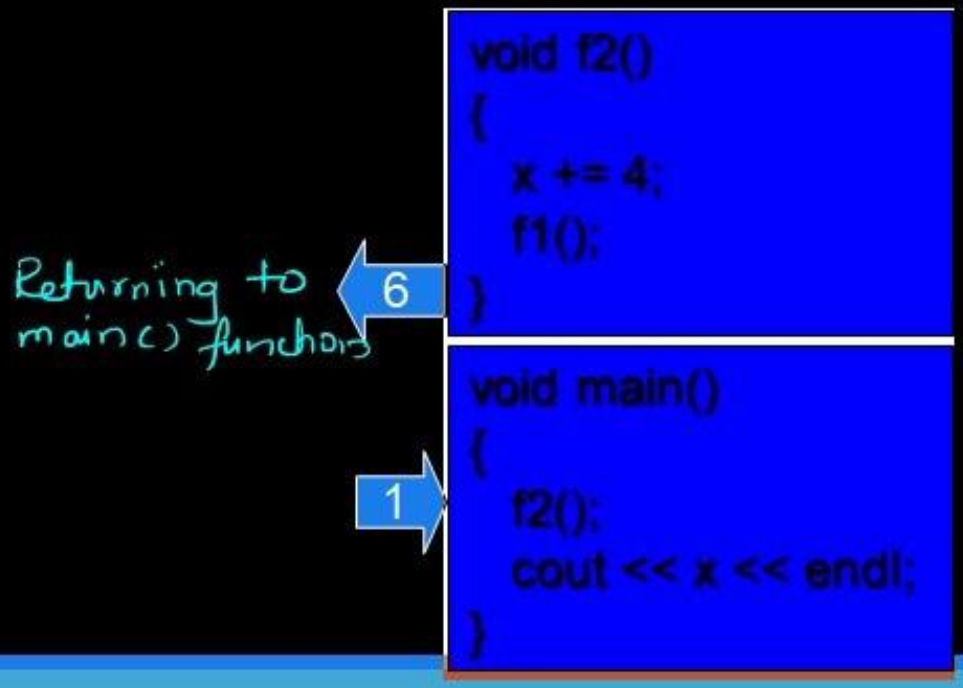
I. Using Global Variables

```
#include <iostream.h>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```



I. Using Global Variables

```
#include <iostream>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```

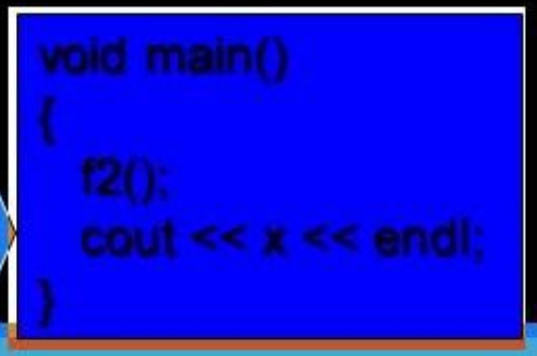


I. Using Global Variables

```
#include <iostream.h>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```

x 5

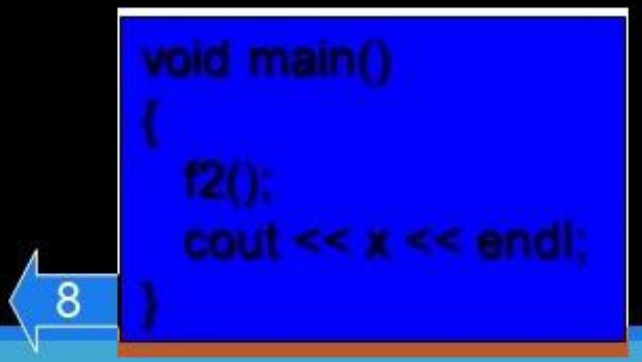
printing the value of x → 7



I. Using Global Variables

```
#include <iostream>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```

x 5



I. Using Global Variables

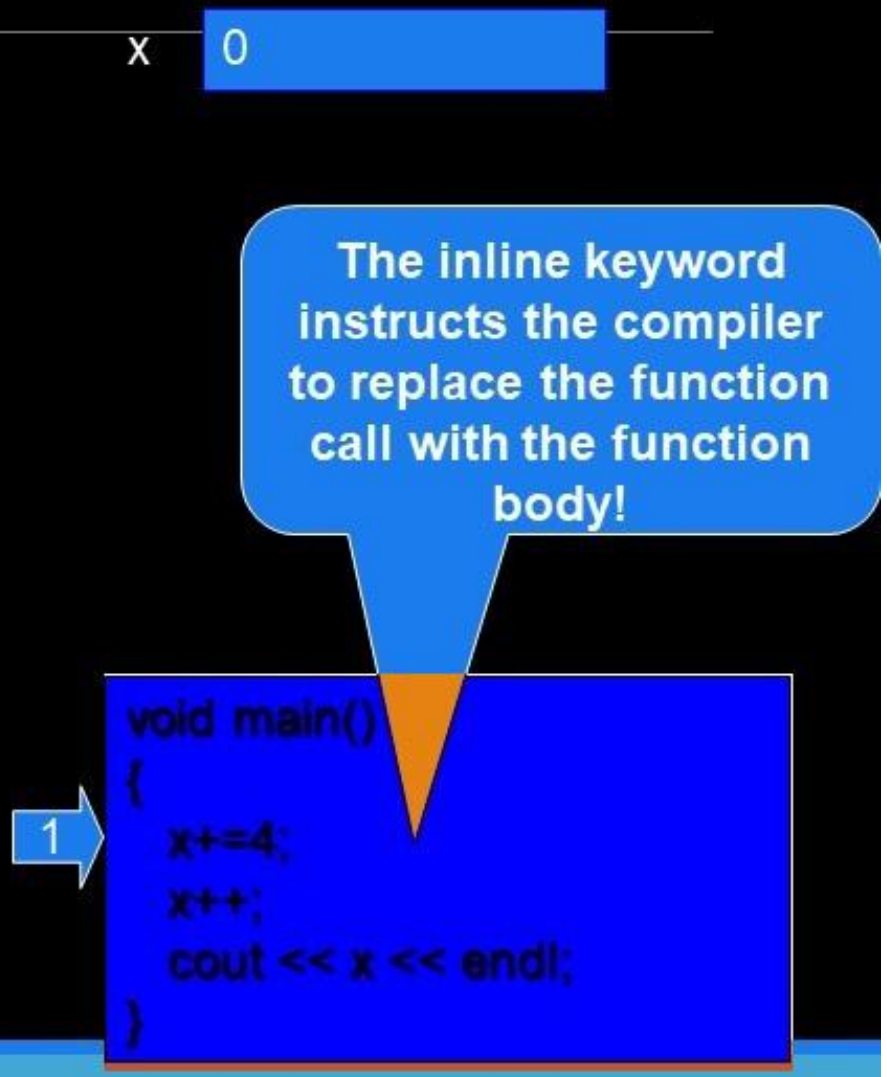
```
#include <iostream>
int x = 0;
void f1() {  x++; }
void f2() { x+=4; f1(); }
void main()
{
  f2();
  cout << x << endl;
}
```

What Happens When We Use Inline Keyword?

```
#include <iostream>
int x = 0;
Inline void f1() {  x++; }
Inline void f2() { x+=4; f1();}
void main()
{
  f2();
  cout << x << endl;
}
```

What Happens When We Use Inline Keyword?

```
#include <iostream.h>
int x = 0;
Inline void f1() {  x++; }
Inline void f2() { x+=4; f1();}
void main()
{
  f2();
  cout << x << endl;
}
```



What Happens When We Use Inline Keyword?

```
#include <iostream>
int x = 0;
Inline void f1() {  x++; }
Inline void f2() { x+=4; f1();}
void main()
{
  f2();
  cout << x << endl;
}
```

x 4

2

```
void main()
{
  x+=4;
  x++;
  cout << x << endl;
}
```

What Happens When We Use Inline Keyword?

```
#include <iostream>
int x = 0;
Inline void f1() {  x++; }
Inline void f2() { x+=4; f1();}
void main()
{
  f2();
  cout << x << endl;
}
```

x 5

3

```
void main()
{
  x+=4;
  x++;
  cout << x << endl;
}
```

What Happens When We Use Inline Keyword?

```
#include <iostream>
int x = 0;
Inline void f1() {  x++; }
Inline void f2() { x+=4; f1();}
void main()
{
  f2();
  cout << x << endl;
}
```

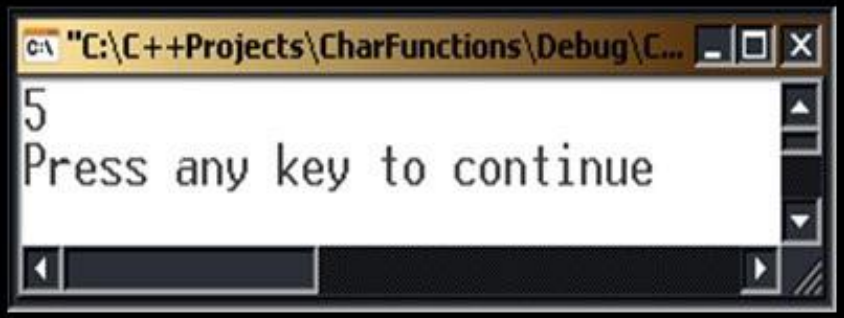
x 5


```
void main()
{
  x+=4;
  x++;
  cout << x << endl;
}
```

4

What Happens When We Use Inline Keyword?

```
#include <iostream>
int x = 0;
Inline void f1() {  x++; }
Inline void f2() { x+=4; f1();}
void main()
{
  f2();
  cout << x << endl;
}
```



Drawback of using Global Variables?

- Not safe! (Data Inconsistency)
  - If two or more programmers are working together in a program, one of them may change the value stored in the global variable without telling the others who may depend in their calculation on the old stored value!
- Against The Principle of Information Hiding!
  - Exposing the global variables to all functions is against the principle of information hiding since this gives all functions the freedom to change the values stored in the global variables at any time (unsafe!)

Local Variables

- Local variables are declared inside the function body and exist as long as the function is running and destroyed when the function exit
- You have to initialize the local variable before using it
- If a function defines a local variable and there was a global variable with the same name, the function uses its local variable instead of using the global variable

Example: Global and Local Variables

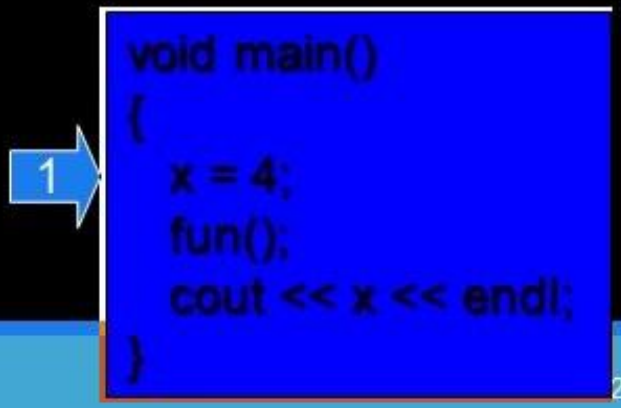
```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature

void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```

Example: Global and Local Variables

```
#include <iostream.h>
int x; // Global variable
void fun(); // function signature
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```

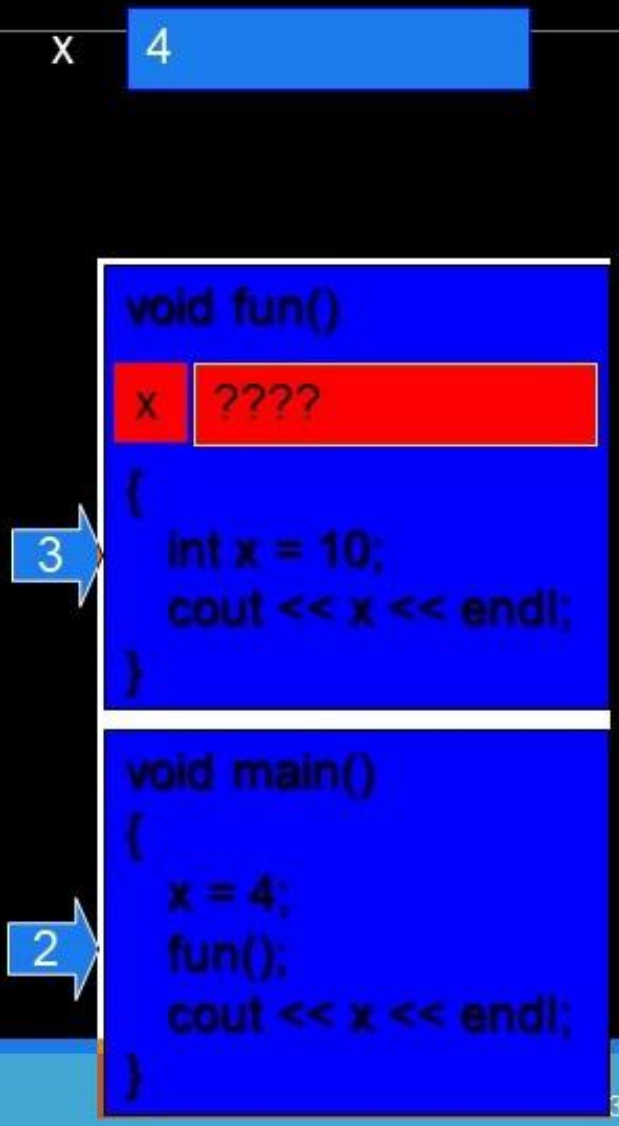
Example: Global and Local Variables

```
#include <iostream.h>
int x; // Global variable
void fun(); // function signature
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```



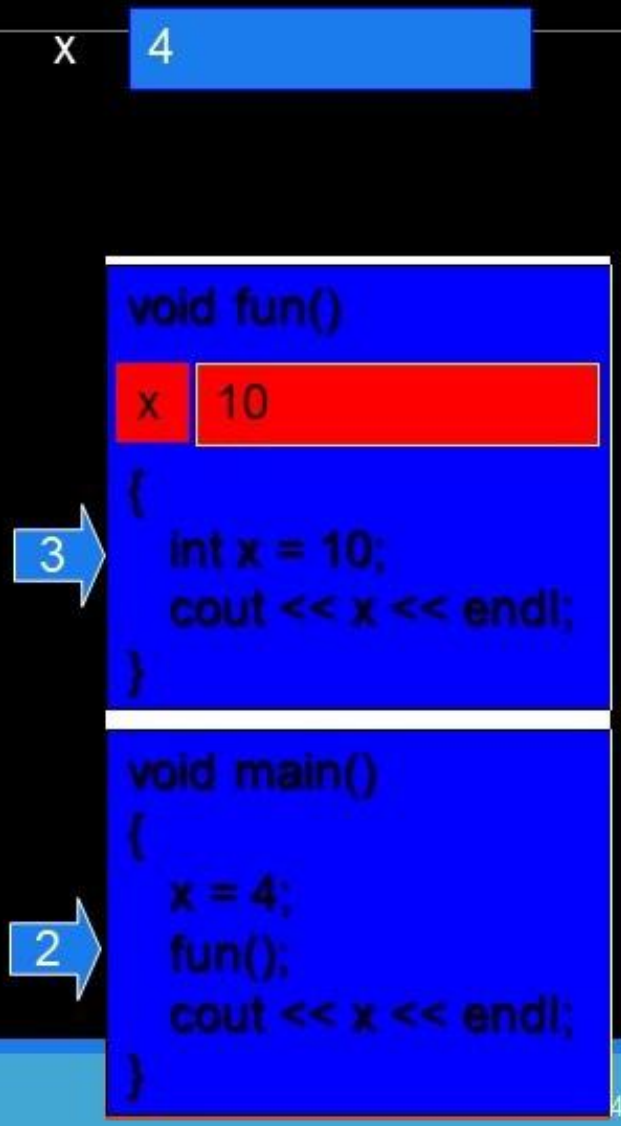
Example of Defining and Using Global and Local Variables

```
#include <iostream.h>
int x; // Global variable
void fun(); // function signature
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```



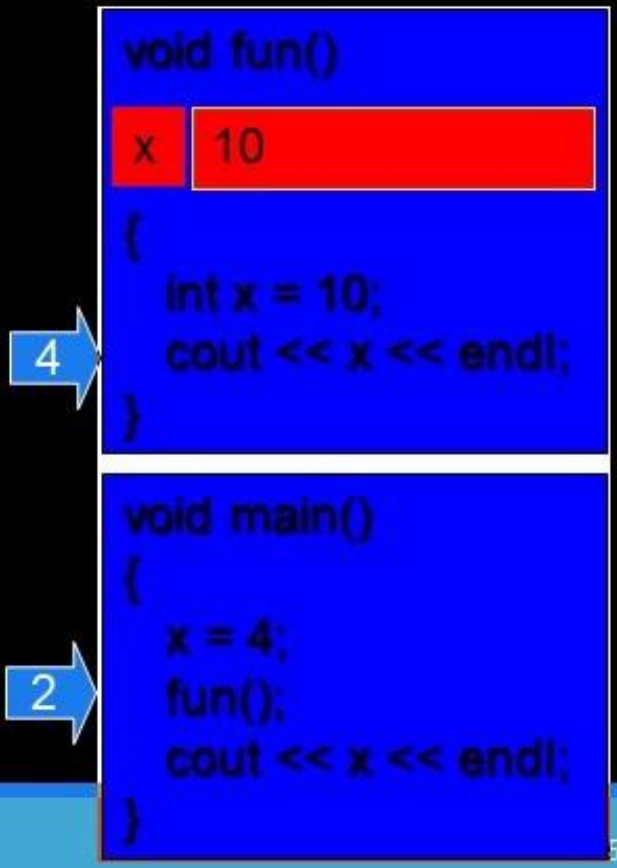
Example of Defining and Using Global and Local Variables

```
#include <iostream.h>
int x; // Global variable
void fun(); // function signature
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```



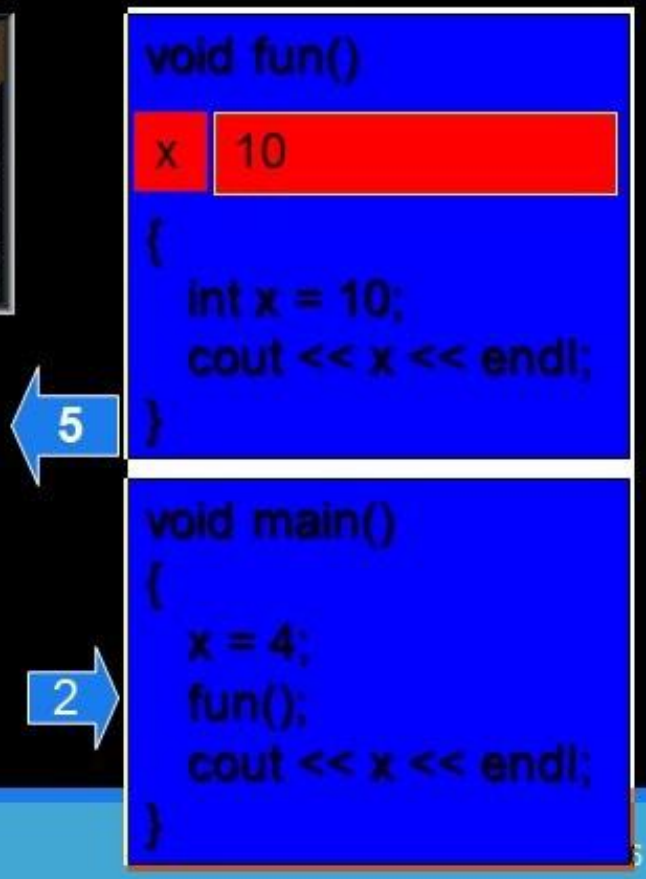
Example of Defining and Using Global and Local Variables

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```



Example of Defining and Using Global and Local Variables

```
#include <iostream.h>
int x; // Global variable
void fun(); // function signature
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```



Example of Defining and Using Global and Local Variables

```
#include <iostream.h>
int x; // Global variable
void fun(); // function signature
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```

x 4

6

```
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
```

Example of Defining and Using Global and Local Variables

```
#include <iostream.h>
int x; // Global variable
Void fun(); // function signature
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
void fun()
{
   int x = 10; // Local variable
   cout << x << endl;
}
```

x 4


```
void main()
{
   x = 4;
   fun();
   cout << x << endl;
}
```

II. Data sharing using Parameters

Function Parameters come in two flavors:

1. ***Value parameters*** – which copy the values of the function arguments
2. ***Reference parameters*** – which refer to the function arguments by other local names and have the ability to change the values of the referenced arguments

Value Parameters (call by value)

- This is what we use to declare in the function signature or function header, e.g.
  - int max (int x, int y);
    - Here, parameters x and y are value parameters
    - When you call the max function as *max(4, 7)*, the values 4 and 7 are copied to x and y respectively
    - When you call the max function as *max (a, b),* where a=40 and b=10, the values 40 and 10 are copied to x and y respectively
    - When you call the max function as *max(a+b, b/2),* the values 50 and 5 are copies to x and y respectively
- Once the value parameters accepted copies of the corresponding arguments data, they act as local variables!

Example of Using Value Parameters and Global Variables

x 0

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
   cout << x << endl;
 x=x+5;
}
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```

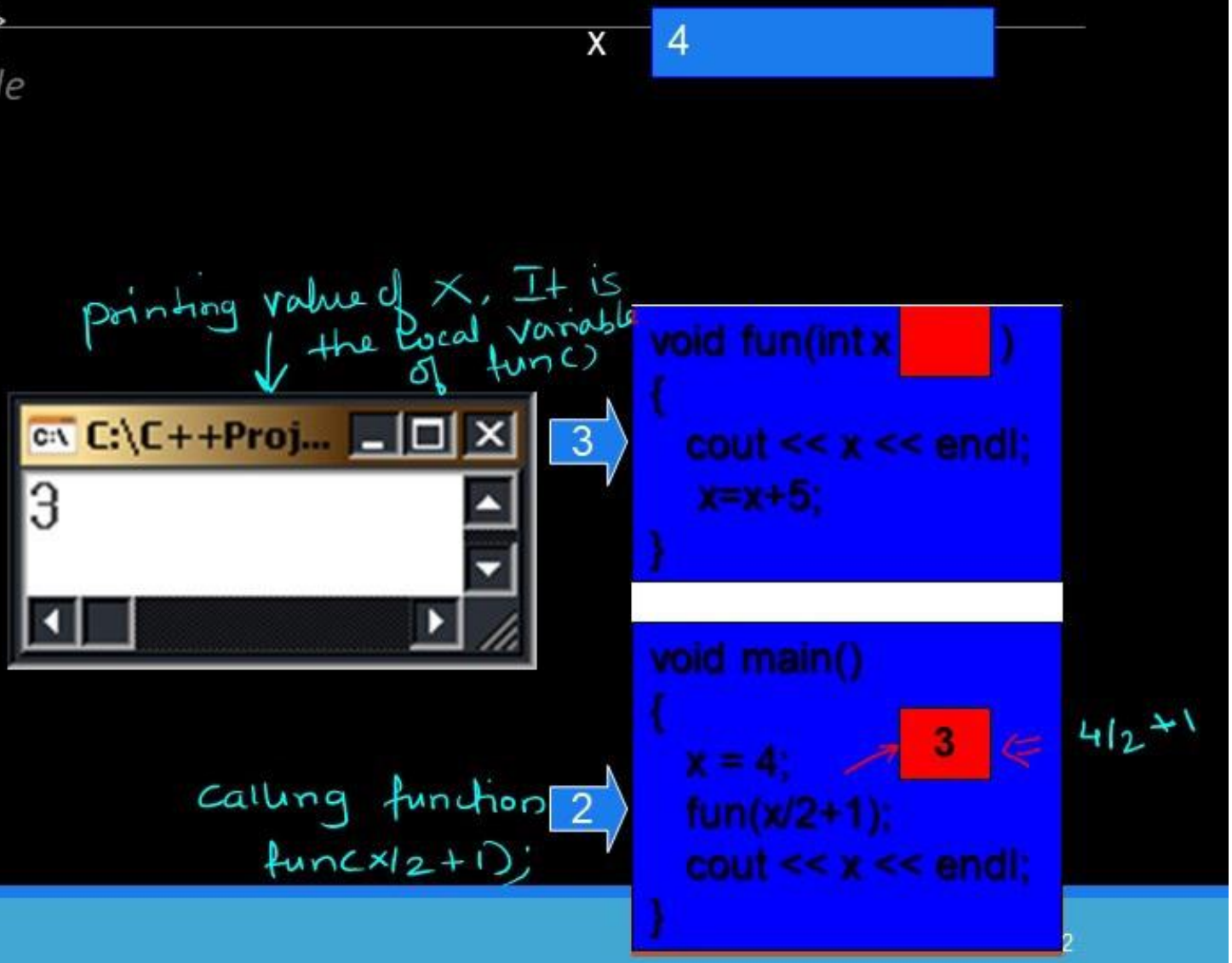
Global variable accessed in main function

1

```
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```

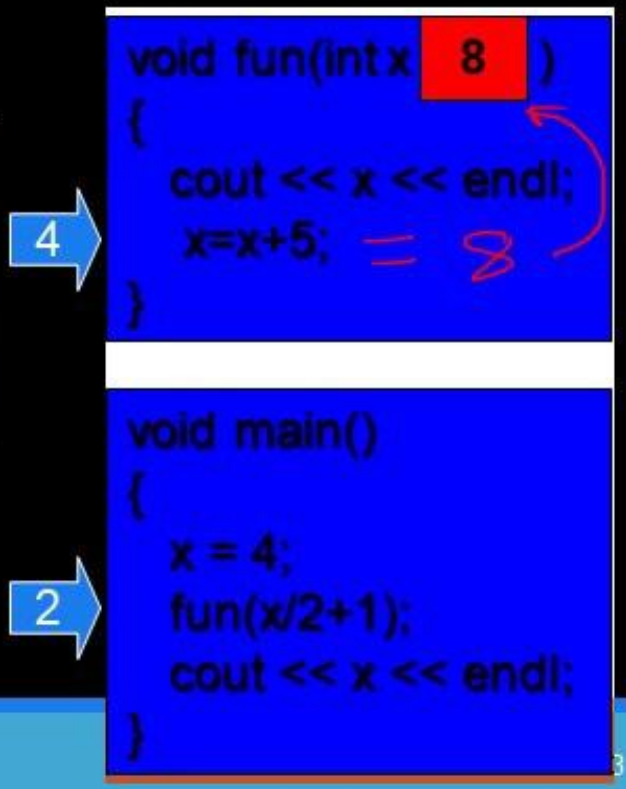
Example of Using Value Parameters and Global Variables

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
   cout << x << endl;
 x=x+5;
}
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```



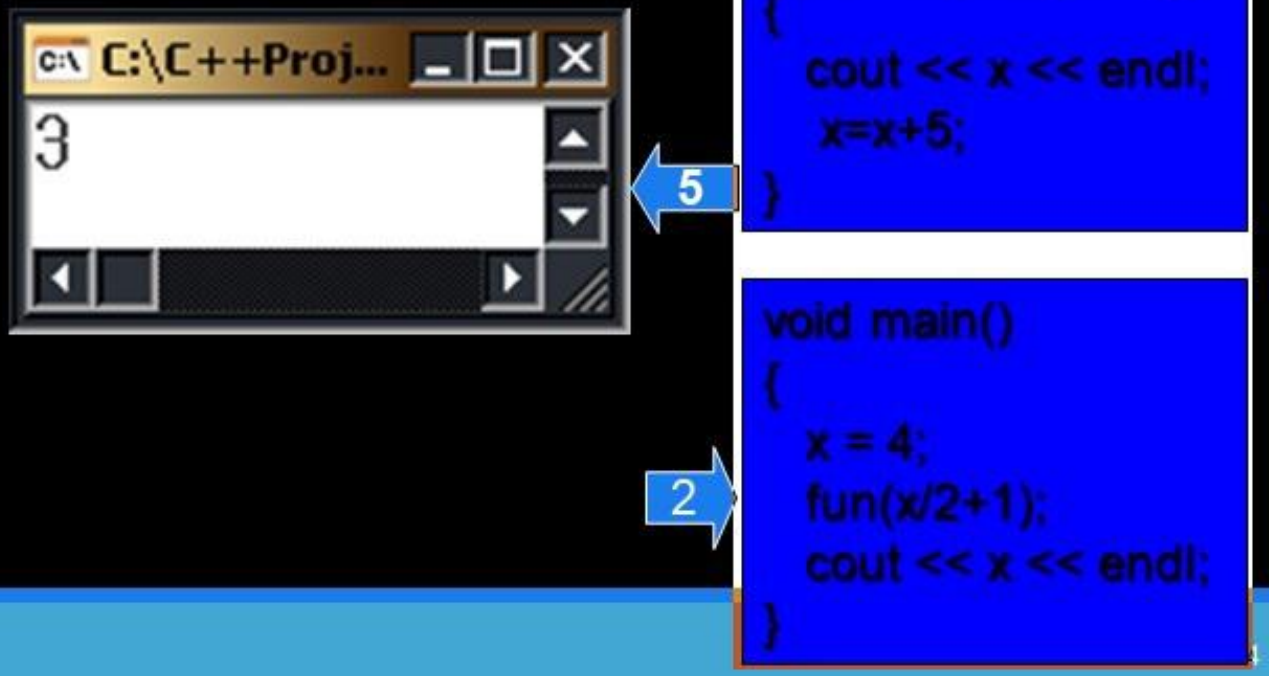
Example of Using Value Parameters and Global Variables

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
   cout << x << endl;
 x=x+5;
}
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```



Example of Using Value Parameters and Global Variables

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
   cout << x << endl;
 x=x+5;
}
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```



Example of Using Value Parameters and Global Variables

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
   cout << x << endl;
 x=x+5;
}
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```

x 4

It point global variable = 4 → 6

```
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```

Example of Using Value Parameters and Global Variables

```
#include <iostream.h>
int x; // Global variable
void fun(int x)
{
   cout << x << endl;
 x=x+5;
}
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```

x 4


```
void main()
{
   x = 4;
   fun(x/2+1);
   cout << x << endl;
}
```

7

Reference Parameters

- As we saw in the last example, any changes in the value parameters don't affect the original function arguments
- Sometimes, we want to change the values of the original function arguments or return with more than one value from the function, in this case we use reference parameters
  - A reference parameter is just another name to the original argument variable
  - We define a reference parameter by adding the & in front of the parameter name, e.g.

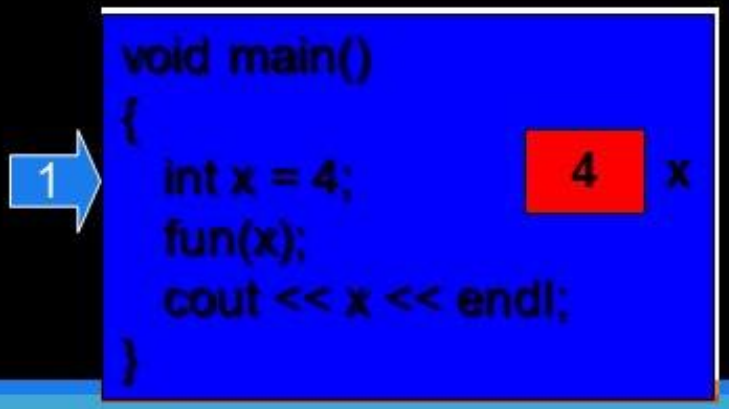
Syntax: double update (double & x);

Example of Reference Parameters

```
#include <iostream.h>
void fun(int &y)
{
  cout << y << endl;
  y=y+5;
}
void main()
{
  int x = 4; // Local variable
  fun(x);
  cout << x << endl;
}
```

reference to x

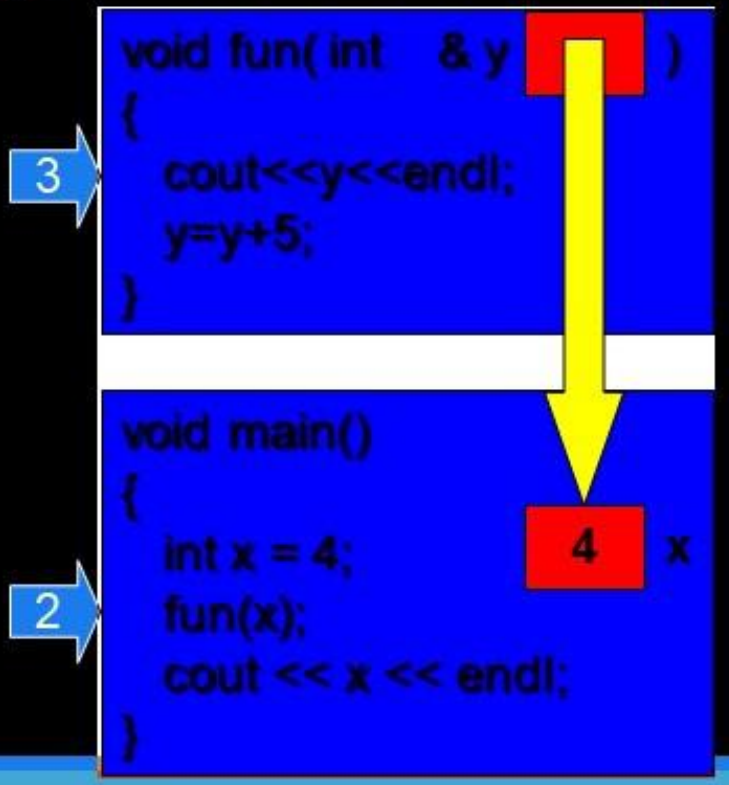
y can directly refer memory location of x. whereas x is the local variable of main() function



Example of Reference Parameters

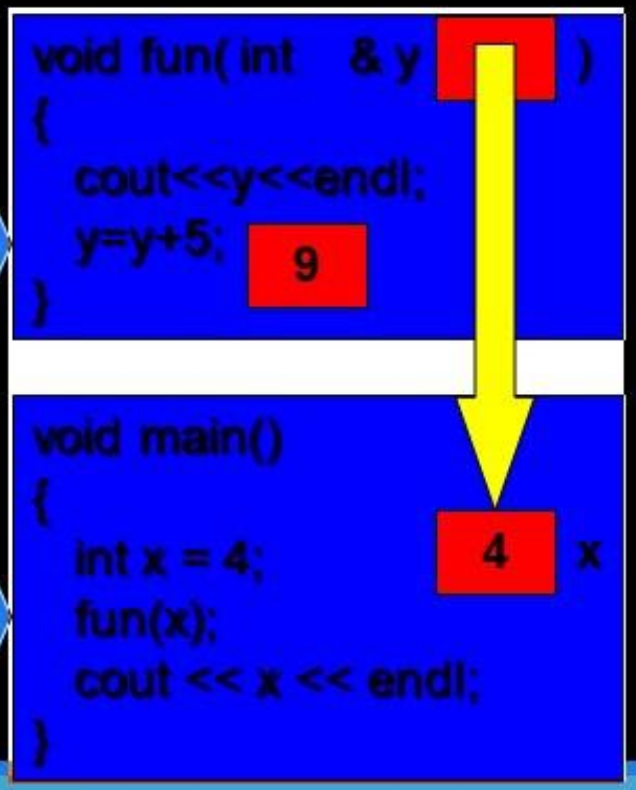
```
#include <iostream.h>
void fun(int &y)
{
   cout << y << endl;
 y=y+5;
}
void main()
{
   int x = 4; // Local variable
   fun(x);
   cout << x << endl;
}
```

Printing value of main() function by fun() function



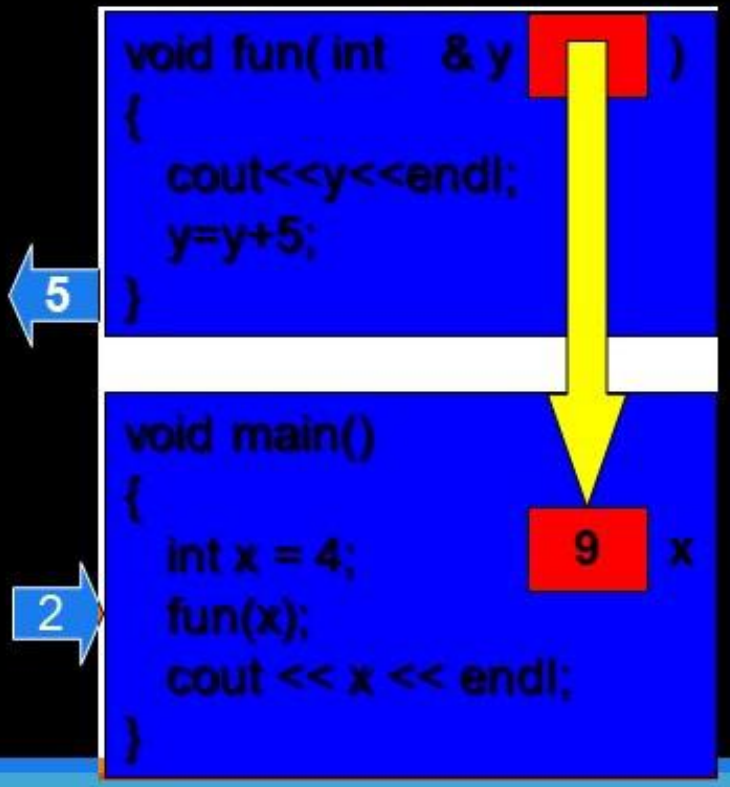
Example of Reference Parameters

```
#include <iostream.h>
void fun(int &y)
{
   cout << y << endl;
 y=y+5;
}
void main()
{
   int x = 4; // Local variable
   fun(x);
   cout << x << endl;
}
```



Example of Reference Parameters

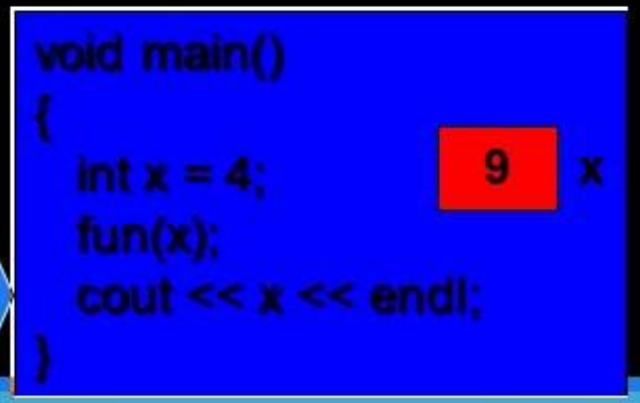
```
#include <iostream.h>
void fun(int &y)
{
   cout << y << endl;
 y=y+5;
}
void main()
{
   int x = 4; // Local variable
   fun(x);
   cout << x << endl;
}
```



Example of Reference Parameters

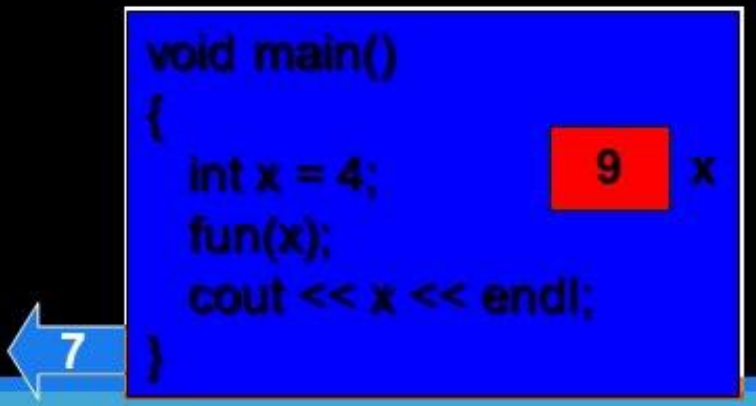
```
#include <iostream.h>
void fun(int &y)
{
   cout << y << endl;
 y=y+5;
}
void main()
{
   int x = 4; // Local variable
   fun(x);
   cout << x << endl;
}
```

printing modified value of x.



Example of Reference Parameters

```
#include <iostream.h>
void fun(int &y)
{
   cout << y << endl;
 y=y+5;
}
void main()
{
   int x = 4; // Local variable
   fun(x);
   cout << x << endl;
}
```



Call by address

```
void swap(int *x, int *y)
{
int z;  z=*x;   *x = *y ;  *y = z;
}
void main()
{
int a= 100 , b = 200;
cout<<  a << b;
swap(&a, &b);
cout<< a << b;
}
```

→ Address of a & b will get stored in pointer variables x & y

Exchange of values done here, It directly swap the values of a & b of main function

→ print 100 200 as values of a & b before calling swap() function

→ print 200 100 as values of a & b after calling swap() function

- Parameters are passed by address to the function
- Address of the memory location copied to the pointer arguments of the called function
- Using pointer arguments local variable of the calling function can be accessed

Topic 5:

NEW AND DELETE:

DYNAMIC MEMORY ALLOCATION OPERATORS

Memory Allocation

There are essentially two types of memory allocation

Static – Done by the compiler automatically (implicitly).

- **Global variables (outside a function)** -- memory is allocated at the start of the program, and freed when program exits; alive throughout program execution
  - can be access anywhere in the program.
- **Local variables (inside a function)** – memory is allocated when the routine starts and freed when the routine returns.
  - A local variable cannot be accessed from another routine.
- Allocation and free are done implicitly.
- No need to explicitly manage memory is nice (easy to work with), but has limitations!
  - Using static allocation, the array size must be fixed.
  - Consider the roster for the class? What is the number of people in the class ? It is fixed.

Memory Allocation

There are essentially two types of memory allocation

- Wouldn't it be nice to be able to have an array whose size can be adjusted depending on needs.
  - Dynamic memory allocation deals with this situation.

Dynamic – Done explicitly by programmer.

- It allocate and de-allocate as much memory as programmer want and also as and when he/she want it
- Programmer explicitly requests the system to allocate memory and return starting address of memory allocated.
- This address can be used by the programmer to access the allocated memory.
- This memory will be allocated at run time (during the execution of the program)
- When done using memory, it must be **explicitly** freed.

The ‘new’ Operator:
Explicitly allocating memory

- “new” operator is used to dynamically allocate memory
- Can be used to allocate memory to a single variable, an array, an object or an array of objects
- The new operator returns pointer to the type allocated
- Before the assignment, the pointer may or may not point to a legitimate memory
- After the assignment, the pointer points to a legitimate memory.
- Examples:

```
char  *my_char_ptr = new  char;
int *my_int_array  = new  int[20];
```

Example:

```
int *ptr = NULL;
ptr = new int();
```

- In the above example, we have declared a pointer variable ‘ptr’ to integer and initialized it to null.
- Then using the “new” operator we allocate memory to the “ptr” variable.
- If memory is available on the heap, the second statement will be successful.
- If no memory is available, then the new operator throws “std::bad_alloc” exception.
- This **NULL** allocation will let programmer know that memory has been allocated successfully or not
- Programmer can use NULL value to check for the same

the 'delete' Operator:
Explicitly freeing memory

- Used to free memory allocated with new operator
- The delete operator should be called on a pointer to dynamically allocated memory when it is no longer needed
- Can delete a single variable/object or an array

  delete PointerName; → delete memory allocated to single variable

  delete [] ArrayName; → delete memory allocated to array
- After delete is called on a memory region, that region should no longer be accessed by the program
- Convention is to set pointer to deleted memory to NULL
  - Any new must have a corresponding delete --- if not, the program has memory leak.
  - New and delete may not be in the same routine.

Example of declaring an array with new operator

```
//************* Example 1:**************//
#include<iostream>
int main()
{
float *pts = NULL;
pts=new float[100];
if(!pts)
{  cout<<"Problem  with Memory  allocation<<endl;  }
else {
    for (int i = 0;i<10;i++)
    cin>>*(pts+i);  // cin>>pts[i];
  }
for (int i = 0;i<10;i++)
            cout<<*(pts+i);
delete[] pts;
}
```

Initially pointer is NULL

Array declaration using new operator

This is to check whether pts not NULL

Reading data using for loop

pointer + index i

deallocate memory

```
//*****************Similar Example:*********************//
#include<iostream>
int main()
 {
  int *ptr = NULL;
  ptr = new int();
  int *var = new int(12);

  if( ! ptr)
  {
     cout<<"bad memory allocation"<<endl;
  }
  else
 {
     cout<<"memory allocated successfully"<<endl;
     *ptr = 10;
     cout<<"*ptr = "<<*ptr<<endl;
     cout<<"*var = "<<*var<<endl;
 }
double *myarray = NULL;
myarray = new double[10];
If (! myarray)
 { cout<<"memory not allocated"<<endl; }
else
  {
   for(int i=0;i<10;i++)
     myarray[i] = i+1;
     cout<<"myarray values : ";
   for(int i=0;i<10;i++)
     cout<<myarray[i]<<"\t";
  }
delete ptr;
delete var;
delete[] myarray;
}
```

The Heap

- Large area of memory controlled by the runtime system that is used to grant dynamic memory requests.
- It is possible to allocate memory and "lose" the pointer to that region without freeing it. This is called a memory leak.
- A memory leak can cause the heap to become full
- If an attempt is made to allocate memory from the heap and there is not enough memory, an exception is generated (error)

sample5.cpp

Why use dynamic memory allocation?

- Allows data (especially arrays) to take on variable sizes (e.g. ask the user how many numbers to store, then generate an array of integers exactly that size).
- Allows locally created variables to live past end of routine.
- Allows us to create many structures used in Data Structures and Algorithms

Topic 6:

FUNCTION WITH DEFAULT ARGUMENTS

Default argument function

- The default arguments are used when you provide no arguments or only few arguments while calling a function.
- A Default value assignment to an argument will be from right to left
- Must be a constant declared in prototype:

  Example:

  ```
  void evenOrOdd(int = 0);  // function prototype
  ```

- Can be declared in header if no prototype used
- Multi-parameter functions may have default arguments for some or all of them

  Example:

  ```
  int getSum(int, int=0, int=0);
  ```

Example:

```
//*******Example 1*********//
#include <iostream>
void display(char *name, int rn, char div = 'E')
{
cout<<"name is:" <<name;
cout<<"roll no is :"<<rn;
cout<<"division is :"<<div;
}
int main()  {
display("xyz",10);
}
```

single default argument

Only two parameters are passed, third argument is default

```
//*********Example  2***********//
#include <iostream>
void display(char *name ="xyz", int rn=12, char div = 'E')
{
cout<<"name is:" <<name;
cout<<"roll no is :"<<rn;
cout<<"division is :"<<div;
}
int main()  {
display();
}
```

All the arguments are default

No parameter is passed to this function

Key points:

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when calling function provides values for them.
  - sum(int x, int y, int z = 0, int w = 0)
  - For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.
- During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15 and 25 to x, y, and z. Therefore, the default value is used for w only.

Key points:

- Once default value is used for an argument in function definition, all subsequent arguments to it must have default value.
- It can also be stated as default arguments are assigned from right to left.
  - For example, the following function definition is invalid as subsequent argument of default variable z is not default.

Example:

```
int sum(int x, int y, int z=0, int w)
```