Competitve Programming using C++

Dr. Pushpalatha K
Professor
Sahyadri College of Engineering & Management

Procedure-Oriented Programming Systems

Procedure Oriented System has the following programming patterns

- ✓ Divides code into functions.
- ✓ Data (contained in structure variables) is passed from one function to another to be read from or written into.
- ✓ Focus is on Procedures or functions.
- ✓ Procedures or functions are dissociated from data & are not a part of it.
  - Instead they receive structure variables or their addresses & work upon them

- ✓ Drawbacks
  - Data is not secure and can be manipulated by any function or procedure.
  - Associated functions that were designed by library programmer don’t have rights to work upon the data.
  - They are not a part of structure definition itself
    - Application program might modify the structure variables by some code inadvertently written in application program itself
  - To detect the faulty code the program need to be debugged which involves the visual inspection of the entire code
    - Debugging is not limited to associated functions only.

Drawbacks contd..

- While distributing the application, application programmer can’t be sure that program would run successfully.
- Every new piece of code accessing structure variable will have to be inspected and tested again to ensure that it doesn’t corrupt the members of structure.
- Compilers that implement procedure oriented programming systems don’t prevent unauthorized functions from accessing or manipulating the structure variables.

✓ Lack of data security of procedure oriented programs has led to **Object Oriented Programming Systems**

Object Oriented Programming Systems

- ✓ Model real-world objects
  - – Real world objects have internal parts and interfaces that enable us to operate them.
    - Eg: Washing Machine
- ✓ If a perfect interface is required to work on an object, it will also have exclusive rights to do so.

- ✓ In OOPS, the associated functions of the data structure can be given exclusive rights to work upon its variables.
  - – all other pieces of code can be prevented from accessing the data members of the variables of this structure.
- ✓ Compilers that implement OOPS enable data security by enforcing the prohibition.
- ✓ OOPS makes sure the data are initialized.
- ✓ OOPS Features –
  - – Inheritance
  - – Polymorphism

Inheritance

- ✓ Inheritance allows one structure to inherit the characteristics of an existing structure.
  - The new structure can contain new data members along with the data members from which new structure has inherited.
  - In Inheritance, both data and functions may be inherited
    - Parent class can be given the general characteristics, while its child may be given more specific characteristics.
  - Inheritance allows code reusability by keeping code in a common place – the base structure.
  - Inheritance allows code extensibility by allowing creation of new structures that are suited to our requirements compared to existing structures.

Polymorphism

- ✓ In OOPS, the functions can be used with different set of formal arguments, but have the same name is known as Polymorphism
- ✓ Polymorphism is of two types: static and dynamic.

Comparison of C++ with C

- ✓ C++ is an extension of C language.
- ✓ It is a proper superset of C language.
  - – a C++ compiler can compile programs written in C language.
    - the reverse is not true.
  - – A C++ compiler can understand all the keywords that a C compiler can understand.
- ✓ Decision-making constructs, looping constructs, structures, functions, etc. are written in exactly the same way in C++ as they are in C language
  - – C++ provides additional keywords and language constructs that enable it to implement the object-oriented paradigm

Console output in C++

- ✓ The output functions in C language, such as printf(), can be included in C++ programs
- ✓ **cout** is an instance of **ostream** class and stands as an alias for the **console output device i.e. monitor**
- ✓ The **cout** is used in conjunction with the stream **insertion operator <<**
  - The value on the right side of the insertion operator is 'inserted' into the stream resulting in displaying the inserted value on the monitor.
- ✓ Since the **cout** and the **<<** have been declared in file **iostream.h**, it needs to be included in the source code.

- Object **endl** allows to insert a new line into the output stream
- Outputting in C++

```
#include<iostream.h>
void main()
{
     int x,y;
    x=10;
    y=20;
    cout<<x;
    cout<<endl;
    cout<<y;
}
```

```
#include<iostream.h>
void main()
{
int rollno;
char section;
float cgpa;
double avgmarks;
char * name;
rollno=1;
section=‘C’;
cgpa=9.5;
avgmarks=550;
name=“Rama”;
cout<<rollno;
cout<<endl;
cout<<section;
cout<<endl;
cout<<cgpa;
cout<<endl;
cout<<avgmarks;
cout<<endl;
cout<<name;
cout<<endl;
}
```

- ✓ Cascading the insertion operator

```
#include<iostream.h>
void main()
{
   int x;
   float y;
   x=10;
   y=2.2;
   cout<<x<<endl<<y;
}
```

- ■ Outputting constants using the insertion operator

```
#include<iostream.h>
void main()
{
   cout<<10<<endl<<"Hello
     World\n"<<3.4;
}
```

Console Input in C++

- ✓ The input functions in C language, such as scanf(), can be included in C++ programs.
- ✓ **Cin is** an instance of **istream** class and stands as an alias for the **console input device i.e. keyboard**
- ✓ The **cin** is used in conjunction with the stream **extraction operator >>**
  - The value on the right side of the **extraction** operator is '*extracted*' from the stream originating from keyboard
- ✓ Since the **cin** and the **>>** have been declared in file **iostream.h**, it needs to be included in the source code.

Examples

```
#include<iostream.h>
void main()
{
    int rollno;
    char section;
    Float cgpa;
    cout<<“Enter Roll No”;
    cin>> rollno;
    cout<<“Enter section: ”;
    cin>> section;
    cout<<“Enter cgpa : ”;
    cin>> cgpa;
    cout<<“You entered: ”<< rollno <<“ ”<< section <<“ ”<< cgpa;
}
```

Cascading the extraction operator

```
#include<iostream.h>
void main()
{
    int x,y;
    cout<<“Enter two numbers\n”;
    cin>>x>>y; //cascading the extraction operator
    cout<<“You entered ”<<x<<“ and ”<<y;
}
```

Variables in C++

- ✓ Variables in C++ can be declared anywhere inside a function

```
void main()
{
    int x;
    x=10;
    cout<<“Value of x= ”<<x<<endl;
    int * iPtr;
    iPtr=&x;
    cout<<“Address of x= ”<<iPtr<<endl;
}
```

Reference Variables in C++

- ✓ A reference variable is a reference for an existing variable.
  - another name for the original variable
- ✓ It shares the memory location with an existing variable
- ✓ All operations performed on the reference are actually performed on the original variable.
- ✓ Must be initialized at the time of declaration.
  - Syntax for declaring a reference variable is as follows:
    - **<data-type> & <ref-var-name>=<existing-var-name>;**
  - Eg:

```
int count = 1;
int &cRef = count;
cRef++;
```

Reading the value of a reference variable

✓ The value of a reference variable can be read in the same way as the value of an ordinary variable is read.

```
#include<iostream.h>
void main()
{
    int x,y;
    x=10;
    int &iRef=x;
    y=iRef;             //same as y=x;
    cout<<y<<endl;
    y++;              //x and iRef unchanged
    cout<<x<<endl<<iRef<<endl<<y<<endl;
    iRef=12;
    cout<<x<<endl<<iRef<<endl<<y<<endl;
}
```

Output

```
10
10
10
11
12
12
11
```

Passing by reference

- ✓ Reference variable can be a function argument and thus change the value of the parameter that is passed to it in the function call.

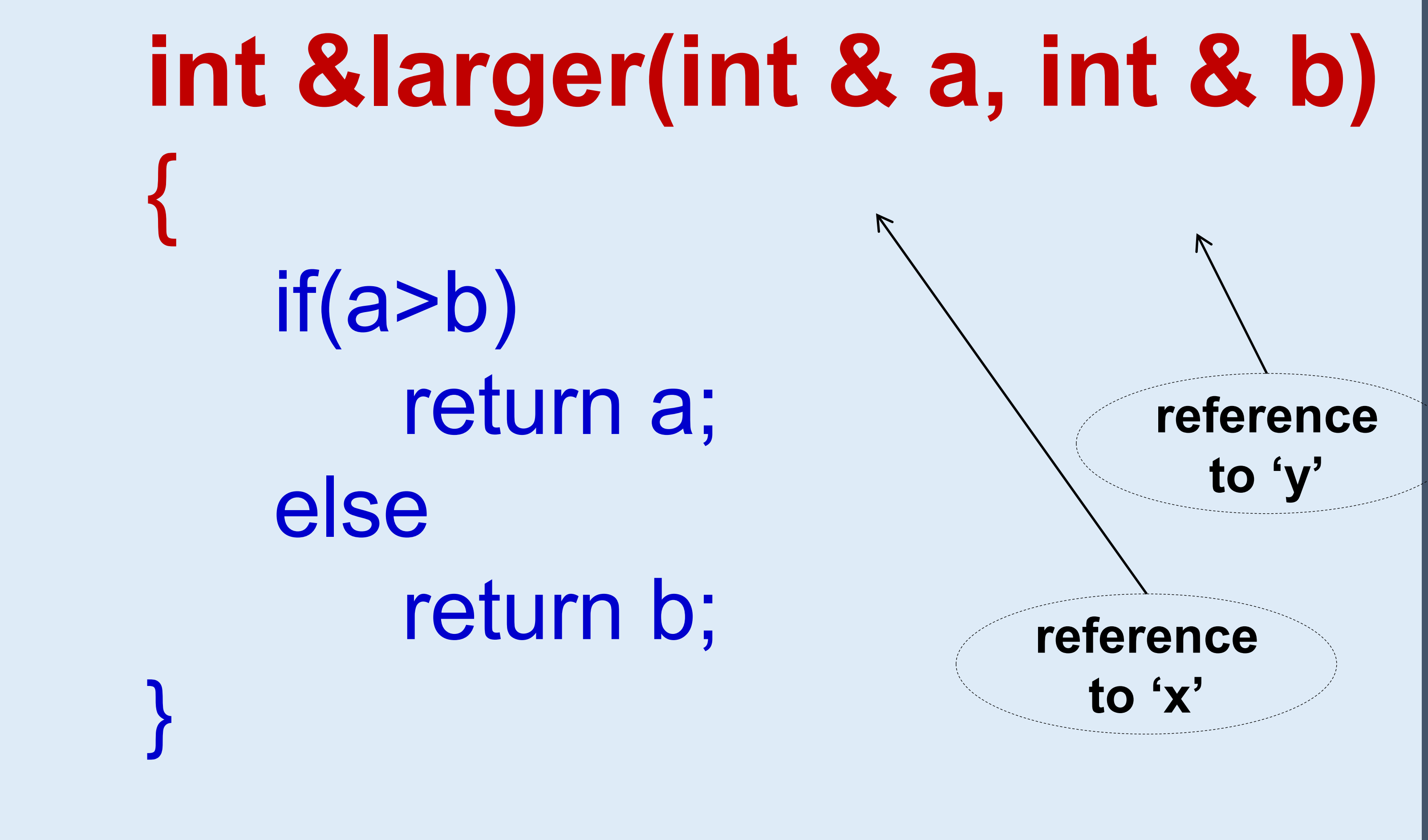
```
#include<iostream.h>
void increment(int &);
void main()
{
    int x;
    x=10;
    increment(x);
    cout<<x<<endl;
}
void increment(int &r)
{
    r++;
}
```

Output

```
11
```

Returning by reference

```
#include<iostream.h>
int & larger(int &, int &);
int main()
{
    int x,y;
    x=10;
    y=20;
    int &r=larger(x,y);
    cout<<x<<endl<<y<<endl;
    r=100;
    cout<<x<<endl<<y<<endl;
}
```



```
int &larger(int & a, int & b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

Output

```
10
100
```

Note: **larger**() function does not return the value 'b' because the return type is **int&** and not **int**. So the address of 'r' becomes equal to the address of 'y'.
Any change in the value of 'r' also changes the value of 'y'.

- ✓ A function that returns by reference returns the address of the returned variable.
- ✓ A call to a function that returns by reference can be placed on the left of the assignment operator
  - the address of the returned variable can be determined from it.

```
#include<iostream.h>
int & larger(int &, int &);
int main()
{
    int x,y;
    x=10;
    y=20;
    larger(x,y)=100;
    cout<<x<<endl<<y<<endl;
}
```

```
int &larger(int & a, int & b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

Output

```
10
100
```

- A call to a function that returns by reference can be placed on the left of the assignment operator
  - ♦ the address of the returned variable can be determined from it.

- ✓ If the compiler finds the name of a non-constant variable on the left of the assignment operator in the source code, it writes instructions in the executable to
  - determine the address of the variable
  - transfer control to the byte that has that address, and
  - write the value on the right of the **assignment** operator into the block that begins with the byte found above
    - Eg: a=10;

- ✓ A function that returns by reference returns the address of the returned variable
- ✓ If the call is found on the left of the assignment operator, the compiler writes necessary instructions in the executable to
  - – transfer control to the byte whose address is returned by the function and
  - – write the value on the right of the assignment operator into the block that begins with the byte found above.
    - Eg: **larger(x,y)=100;**

- ✓ The name of a variable can be placed on the right of the assignment operator.
- ✓ A call to a function that returns by reference can be placed on the right of the assignment operator
- ✓ If the compiler finds the name of a variable on the right of the assignment operator in the source code, it writes instructions in the executable to
  - determine the address of the variable,
  - transfer control to the byte that has that address,
  - read the value from the block that begins with the byte found above, and push the read value into the stack.

  Eg: int b=10;

  a=b;

- A function that returns by reference primarily returns the address of the returned variable
- If the call is found on the right of the assignment operator, the compiler writes necessary instructions in the executable to
  - transfer control to the byte whose address is returned by the function,
  - read the value from the block that begins with the byte found above, and
  - push the read value into the stack.
    - **int &r=larger(x,y);**

- ✓ A constant cannot be placed on the left of the assignment operator.
  - constants do not have a valid address
- ✓ Functions that return by value, return the value of the returned variable, which is a constant.
- ✓ So, a call to a function that returns by value cannot be placed on the left of the **assignment** operator.

✓ Returning the reference of a local variable

- Need to avoid returning a reference to a local variable

```
#include<iostream.h>
int & abc();
void main()
{
    abc()=-1;
}
int &abc()
{
    int x;
    return x;          //returning reference of a local variable
}
```

Function Prototyping

- ✓ Function prototyping is necessary in C++.
- ✓ A prototype describes the function’s interface to the compiler.
- ✓ It tells the compiler the return type of the function as well as the number, type, and sequence of its formal arguments.
- ✓ Syntax: **return_type function_name(argument_list);**
  - Eg: int sum(int,int);
    - indicates that the sum() function returns a value of integer type and takes two parameters both of integer type.
- ✓ By making prototyping necessary, the compiler ensures that
  - the return value of a function is handled correctly
  - Correct number and type of arguments are passed to a function

- ✓ In the absence of prototypes, the compiler will have to assume the type of the returned value.
- ✓ If the function returns the type which is different from the assumed type, then the compiler displays the error.
- ✓ However, if the function definition is in a different file to be compiled separately, then no compile-time errors will arise.
  - Instead, wrong results will arise during run time
- ✓ Thus, function prototyping guarantees protection from errors arising out of incorrect function calls.
- ✓ Function prototyping produces automatic-type conversion wherever appropriate

```
#include<iostream.h>
int add(int,int);   //function prototype
void main()
{
    int x,y,z;
    cout<<“Enter a number: ”;
    cin>>x;
    cout<<“Enter another number: ”;
    cin>>y;
    z=sum(x,y); //function call
    cout<<z<<endl;
}
```

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

```
/*Beginning of def.c*/
struct abc
{
    char a;
    int b;
    float c;
};
struct abc test()
{
    struct abc a1;
    a1.a=‘x’;
    a1.b=10;
    a1.c=1.1;
    return a1;
}
/*End of def.c*/
```

```
/*Beginning of driver.c*/
void main()
{
int x;
x=test(); //no compile time error!!
printf(“%d”,x);
}
/*End of driver.c*/
```

Function Overloading

- ✓ C++ allows two or more functions to have the same name but with different signatures
  - Signature of a function means the number, type, and sequence of formal arguments of the function
- ✓ The compiler differentiates the function depending on their signatures.
- ✓ Hence, the function prototypes should be provided to the compiler for matching the function calls

```
#include<iostream.h>
int add(int,int); //first prototype
int add(int,int,int); //second prototype
void main()
{
    int x,y;
    x=add(10,20); //matches first prototype
    y=add(30,40,50); //matches second prototype
    cout<<x<<endl<<y<<endl;
}
```

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

- The compiler decides which function is to be called based upon the number, type, and sequence of parameters that are passed to the function call
- Since function prototyping is mandatory in C++, it is possible for the compiler to support function overloading properly
- Function overloading is also known as ***function polymorphism***
- Function polymorphism is static in nature because the function definition to be executed is selected by the compiler during compile time itself.

Default Values for Formal Arguments of Functions

- ✓ It is possible to specify default values for some or all of the formal arguments of a function.
  - If no value is passed for an argument when the function is called, the default value specified for it is passed.
  - If parameters are passed in the normal fashion for such an argument, the default value is ignored.
- ✓ Default values can be assigned to more than one argument
  - **int** add(**int**,**int** b=0,**int** c=0);
- ✓ Default values can be given to arguments of any data type
  - int add(int,int,int c=10);
  - double hra(double,double=0.3);
  - void print(char='a');

```
#include<iostream.h>
int add(int,int,int c=10); //third argument has default value
void main()
{
    int x,y;
    x=add(10,20,30); //default value ignored
    y=add(40,50); //default value taken for the third parameter
    cout<<x<<endl<<y<<endl;
}
int add(int a,int b,int c)
{
    return (a+b+c);
}
```

- ✓ There is no need to provide names to the arguments taking default values in the function prototypes.
  - **int add(int,int=0,int=0);**
- ✓ Default values must be specified in function prototypes
  - They should not be specified in the function definitions
- ✓ If default values are specified for the arguments of a function, the function behaves like an overloaded function
  - **Care should be taken when overloading** otherwise ambiguity errors might be caused
    - **Eg: int add(int,int,int=0);**
      int add(int,int);

Inline Functions

- ✓ Inline functions are used to increase the speed of execution of the executable files.
- ✓ C++ inserts calls to the normal functions and the inline functions in different ways in an executable
  - After compiling the various source codes and linking them, a set of machine language instructions called executable program is created.
  - OS loads these instructions into the computer’s memory.
    - Thus, each instruction has a particular memory address.
  - The computer goes through these instructions one by one.

- If there are any instructions to branch out or loop, the control skips over instructions and jumps backward or forward as needed.
- When a program reaches the function call instruction,
  - It stores the memory address of the instruction immediately following the function call.
  - It then jumps to the beginning of the function, executes the function code, and jumps back to the instruction whose address it had saved earlier.

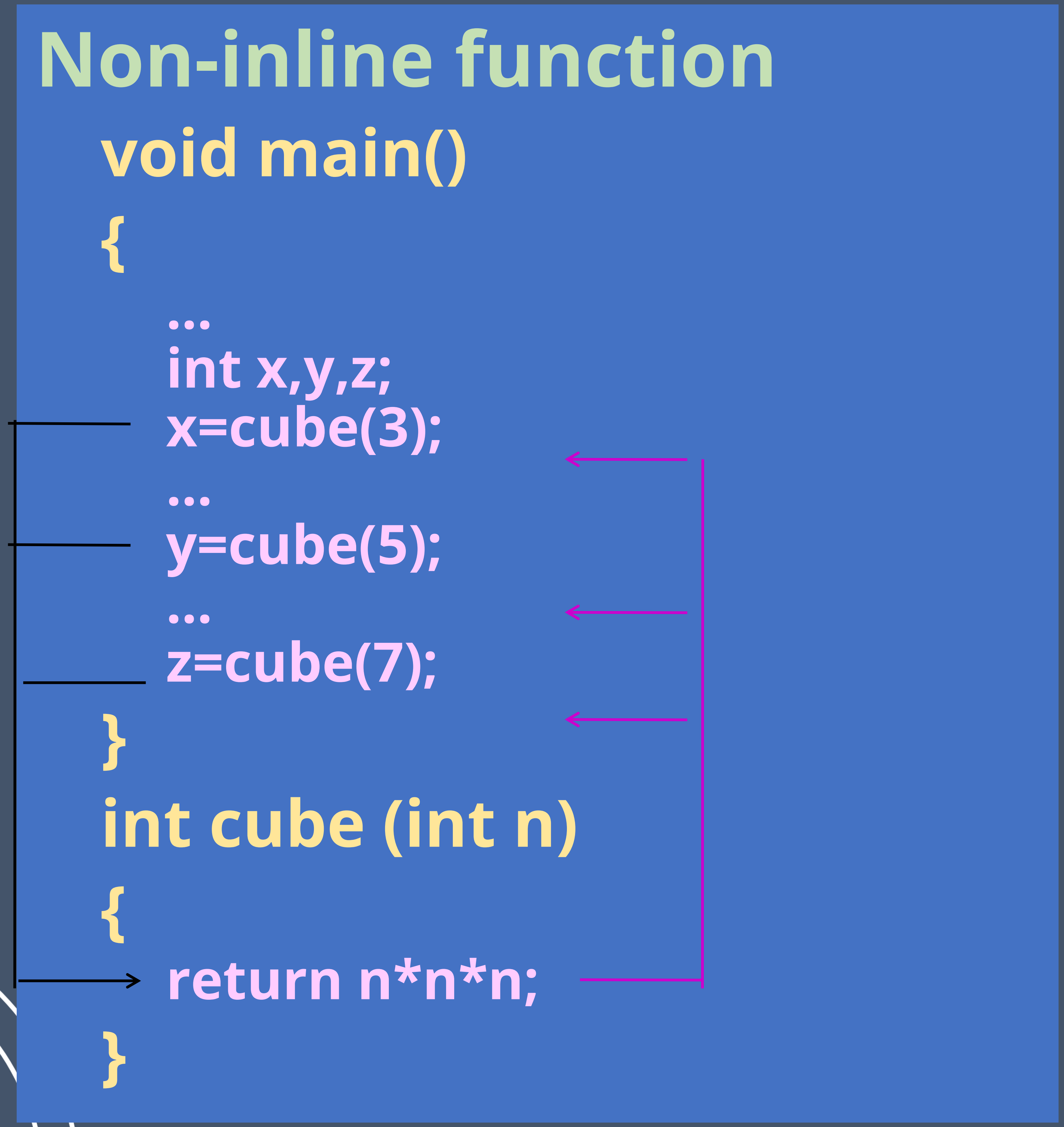
✓ This procedure involves the following overhead

- making the control jump back and forth
- storing the address of the instruction to which the control should jump after the function terminates.

- ✓ **An inline function is a function whose compiled code is ‘in line’ with the rest of the program**
- ✓ The compiler replaces the function call with the corresponding function code.
- ✓ With inline code, the program does not jump to another location to execute the code and then jump back.
- ✓ Advantages:
  - Inline functions, run a little faster than regular functions.

- ✓ Drawbacks:
  - If an inline function is called repeatedly, then multiple copies of the function definition appear in the code
  - The executable program itself becomes so large and occupies a lot of space in the computer’s memory during run time
- ✓ Specifying an inline function:
  - Prefix the definition of the function with the **inline** keyword
  - Define the function before all functions that call it, that is, define it in the header file itself.

Non-inline function



Non-inline function

```
void main()
{
    …
    int x,y,z;
    x=cube(3);
    …
    y=cube(5);
    …
    z=cube(7);
}
int cube (int n)
{
    return n*n*n;
}
```

Inline function

```
void main()
{
    …
    int x,y,z;
    {
        int n;
        n=3;
        x= n*n*n;
    }
    …
    {
        int n;
        n=5;
        y= n*n*n;
    }
    …
    {
        int n;
        n=5;
        z= n*n*n;
    }
    …
}
```

Example

```
#include<iostream.h>
inline double cube(double x) { return x*x*x; }
void main()
{
    double a,b;
    double c=13.0;
    a=cube(5.0);
    b=cube(4.5+7.5);
    cout<<a<<endl;
    cout<<b<<endl;
    cout<<cube(c++)<<endl;
    cout<<c<<endl;
}
```

- ✓ Under some circumstances, the compiler, may not expand the function inline.
  - It will issue a warning that the function could not be expanded inline and then compile all calls to such functions in the normal way.
  - Those conditions are:
    - The function is recursive.
    - There are looping constructs in the function.
    - There are static variables in the function

Inline functions vs Macros in C

- ✓ **Macro**

  #define CUBE(X) X*X*X

  a=CUBE(5.0); **//replaced by a=5.0*5.0*5.0**;

  b=CUBE(4.5+7.5); **//replaced by b=4.5+7.5*4.5+7.5*4.5+7.5**;

  c=CUBE(x++); **//replaced by c=x++*x++*x++;**

- ✓ **#define CUBE(X) X*X*X** may result in wrong answer.
  - Correct definition: **#define CUBE(X) (X)*(X)*(X)**
- ✓ Inline function evaluates the argument only once. Macro evaluates the argument each time it is used in the code.
  - Eg: x=5;
  - CUBE (x++)=?
  - CUBE(++x)=?
- ✓ Note: It is advisable to use inline functions instead of macros.
  - CUBE(x++) undesirably increments ‘x’ thrice

Comparison of C++ with C

C	C++
C compiler cannot execute C++ programs	As C++ is an extension of C language, C++ compiler can execute C programs
2. In C, inclusion of function prototypes is not mandotory	2. In C++, inclusion of function prototypes is mandotory
3. C doesn’t allow for default arguments	3. C++ lets you to specify default arguments in function prototype
4. Declaration of the variables must be at the beginning	4. Declaration of the variables can be anywhere before using

Comparison of C++ with C…

If a C program uses a Local variable that has Same name as global variable, then C uses the value of a local variable	In C++, it is possible to instruct program to use value of global variable with scope resolution
Function overloading doesnot exists.	Function overloading exists.
Function inside the structure is not allowed	Function inside the structure is allowed
Object initialization doesn’t exist	Object initialization (constructor) exist
Data hiding, data abstraction and data encapsulation feature doesn’t exist	Data hiding, data abstraction and data encapsulation exists in C++

Difference between Procedural Programming and Object Oriented Programming

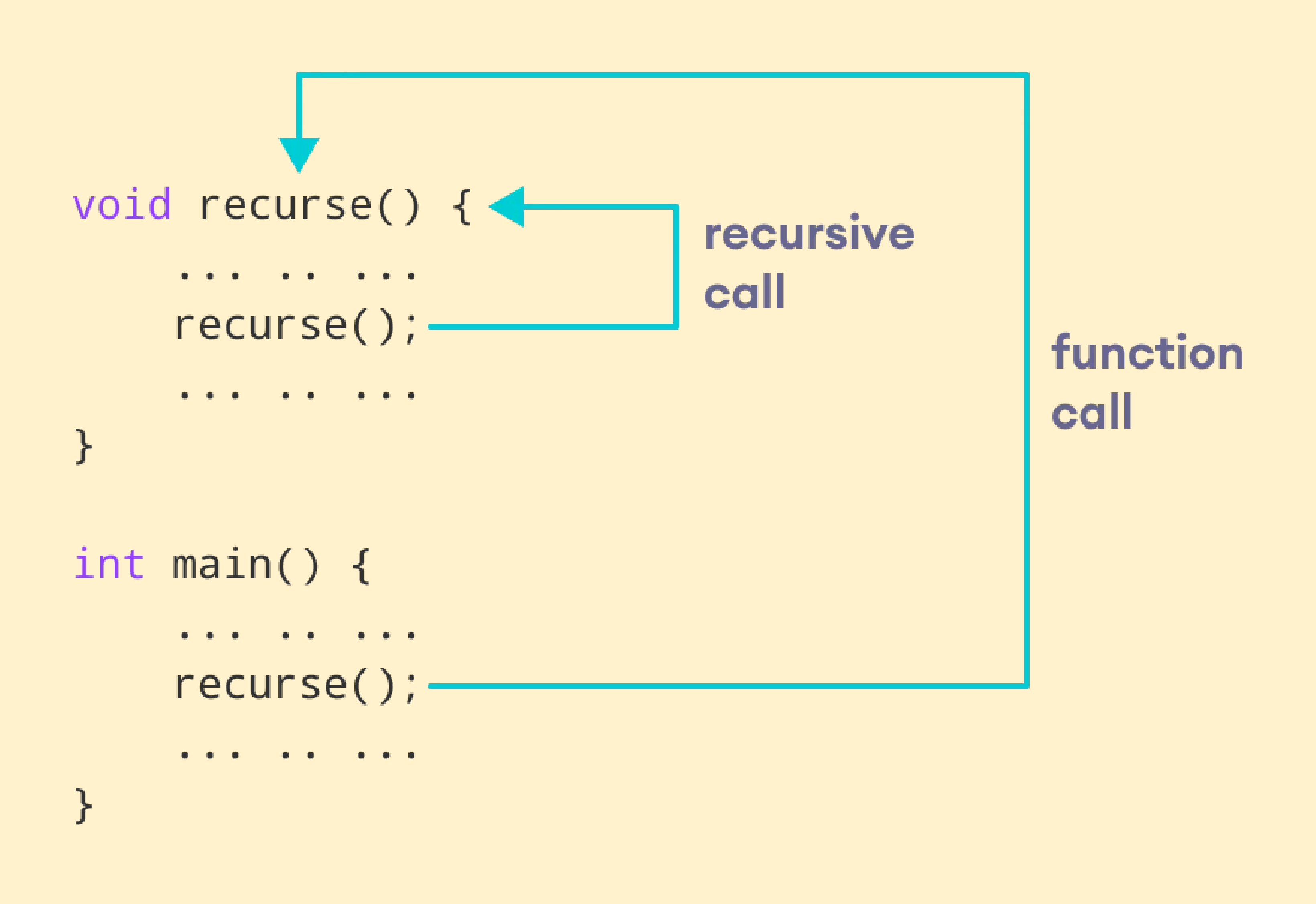
Procedural Oriented Programming	Object Oriented Programming
In procedural programming, program is divided into small parts called functions.	In object oriented programming, program is divided into small parts called objects.
Procedural programming follows top down approach.	Object oriented programming follows bottom up approach.
There is no access specifier in procedural programming.	Object oriented programming have access specifiers like private, public, protected etc.
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way for hiding data so it is less secure.	Object oriented programming provides data hiding so it is more secure.
In procedural programming, overloading is not possible.	Overloading is possible in object oriented programming.
In procedural programming, function is more important than data.	In object oriented programming, data is more important than function.
Procedural programming is based on unreal world.	Object oriented programming is based on real world.
Examples: C, FORTRAN, Pascal, Basic etc.	Examples: C++, Java, Python, C# etc

Pointers vs References in C++

POINTERS	REFERENCES
A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with * operator to access the memory location it points to. Initializton: **int a = 10; int *p = &a;** **OR** **int *p; p = &a;**	A reference variable is an alias, that is, another name for an already existing variable **int a=10; int &p=a;** **int &p; p=a; //incorrect**
A pointer can be re-assigned **p = &b;**	a reference cannot be re-assigned, and must be assigned at initialization **p = &b;** // invalid
A pointer has its own memory address and size on the stack	a reference shares the same memory address (with the original variable) but also takes up some space on the stack.
Pointer can be assigned NULL directly,	reference can not be assigned NULL directly

Recursive Algorithms

✓ A recursive algorithm calls itself with smaller input values and returns the result for the current input by carrying out basic operations on the returned value for the smaller input.



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

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

Example 1: Factorial of a Number Using Recursion

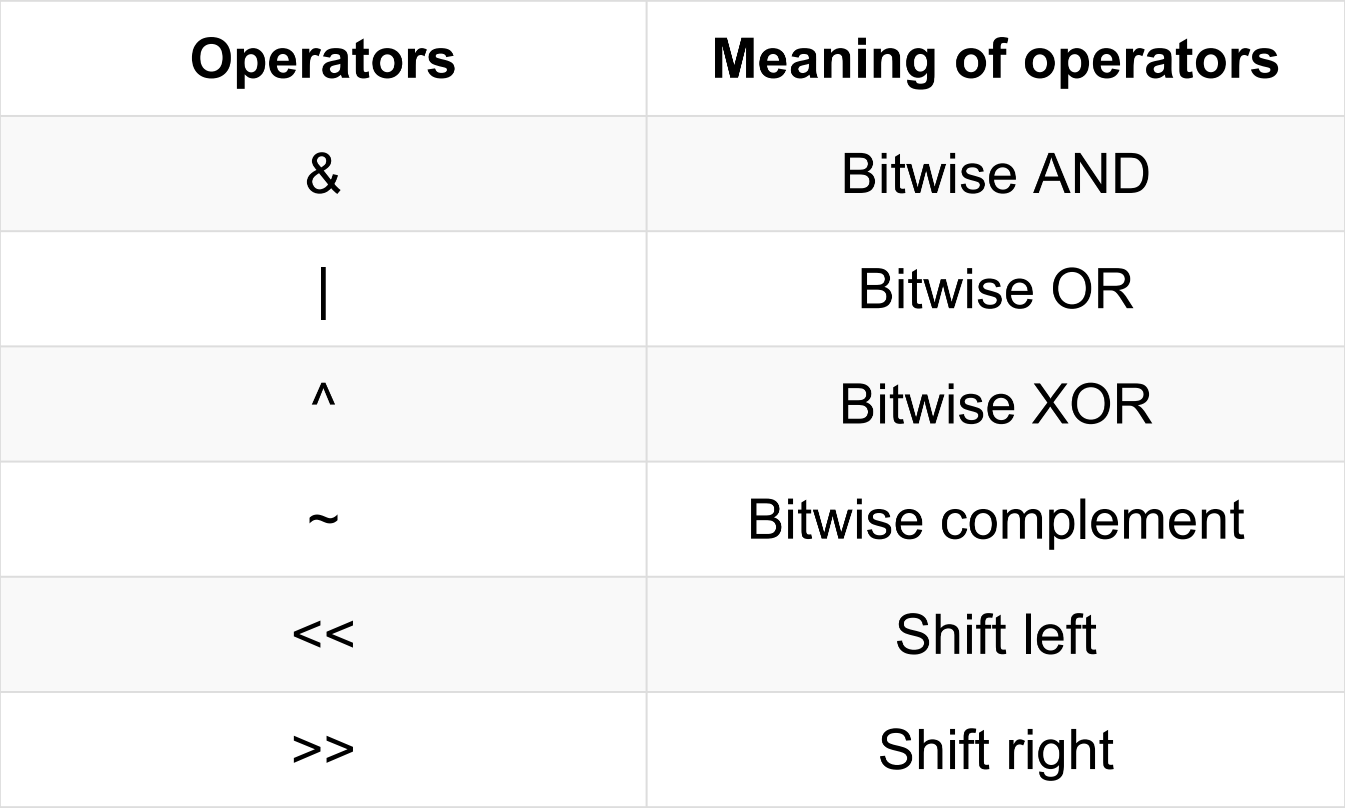
```
#include <iostream>
using namespace std;
int factorial(int);
int main() {
    int n, result;
    cout << "Enter a non-negative number: ";
    cin >> n;
    result = factorial(n);
    cout << "Factorial of " << n << " = " << result;
    return 0;
}
int factorial(int n) {
    if (n > 1) {
        return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

Problems on recursion

- ✓ Program To Calculate Number Power Using Recursion In C++
- ✓ Reverse A Number Using Recursion
- ✓ Print the Fibonacci series using recursion.

Bit manipulation

- ✓ Bit manipulation is the act of algorithmically manipulating bits or other pieces of data
- ✓ Computer programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization.
- ✓ Source code that does bit manipulation makes use of the bitwise operations: AND, OR, XOR, NOT, and bit shifts.



Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise complement
<<	Shift left
>>	Shift right

Check if an integer is even or odd

- ✓ The expression n & 1 returns value 1 or 0 depending upon whether n is odd or even.

```
00010100   &            (n = 20)
00000001               (1)
~~~~~~~~~
00000000
00010101   &            (n = 21)
00000001               (1)
~~~~~~~~~
00000001
```

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

int main()
{
    int n = 5;
     if (n & 1) {
        cout << n << " is odd";
    }
    else {
        cout << n << " is even";
    }
     return 0;
}
```

Detect if two integers have opposite signs or not

- ✓ The expression output x ^ y is negative if x and y have opposite signs.

```
00…000100   ^        (x = 4)
00…001000           (y = 8)
~~~~~~~~~~
00…001100           positive number


00…000100   ^        (x = 4)
11…111000           (y = -8)
~~~~~~~~~~
11…111100           negative number
```

```
#include <iostream>
#include <bitset>
using namespace std;
 int main()
{
    int x = 4;
    int y = -8;
     cout << x << " in binary is " << bitset<32>(x) << endl;
    cout << y << " in binary is " << bitset<32>(y) << endl;
   bool isOpposite = ((x ^ y) < 0);
     if (isOpposite) {
        cout << x << " and " << y << " have opposite signs";
    }
    else {
        cout << x << " and " << y << " don't have opposite signs";
    }
     return 0;
}
```

Swap two numbers without using any third variable

- ✓ use XOR operators to swap two numbers by their property x ^ x = 0

```
#include <iostream>
using namespace std;
 void swap(int &x, int &y)
{
    if (x != y)
    {
        x = x ^ y;
        y = x ^ y;
        x = x ^ y;
    }
}

int main()
{
    int x = 3, y = 4;
     cout << "Before swap: x = " << x << " and y = " << y;
    swap(x, y);
    cout << "\nAfter swap: x = " << x << " and y = " << y;
     return 0;
}
```

Problems on Bit Manupulation

1. Check if a positive integer is a power of 2 without using any branching or loop.
2. Find the position of the rightmost set bit

Dynamic Array

- ✓ A dynamic array is similar to a regular array, but its size is modifiable during program runtime.
- ✓ Dynamic array elements occupy a contiguous block of memory.
- ✓ In C++, a dynamic array can be created using **new** keyword and can be deleted it by using **delete** keyword.

```
#include<iostream>
using namespace std;
int main() {
   int i,n;
   cout<<"Enter total number of elements:"<<"\n";
   cin>>n;
   int *a = new int(n);
   cout<<"Enter "<<n<<" elements"<<endl;
   for(i = 0;i<n;i++) {
      cin>>a[i];
   }
   cout<<"Entered elements are: ";
   for(i = 0;i<n;i++) {
      cout<<a[i]<<" ";
   }
   cout<<endl;
   delete (a);
   return 0;
}
```

Problems on Dynamic Arrays

- ✓ Create a dynamic array and perform the following operations.
  - **Add(x)** - add "x" to the dynamic array of numbers.
  - **Delete(x)** - Delete one occurrence of "x" from the array.
  - **smallest(x)** - print the 1-based xth smallest element in the array.
  - **gcd(L,R)** - print the greatest common divisor of all numbers which have occurred between "L" and "R" (both inclusive) in the array.
    - In delete(x) operation, element "x" is always present in the array. In smallest(x) operation , total number of numbers in the array >= x In gcd(L,R) operation, there will be at least two integers in the range "L" and "R".

The C++ Standard Template Library (STL)

- ✓ The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- ✓ It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized.
  - Working knowledge of template classes is a prerequisite for working with STL.

- ✓ STL has 4 components:
  - Algorithms
  - Containers
  - Functors
  - Iterators

✓ Containers

- Containers are used to manage collections of objects of a certain kind.
  - Sequence Containers: implement data structures that can be accessed in a sequential manner.
    - vector
    - list
    - deque
    - arrays
  - Container Adaptors: provide a different interface for sequential containers.
    - queue
    - priority_queue
    - stack
  - Associative Containers: implement sorted data structures that can be quickly searched
    - set
    - multiset
    - map
    - multimap
  - Unordered Associative Containers: implement unordered data structures that can be quickly searched
    - unordered_set

- ✓ Algorithms
  - The header algorithm defines a collection of functions specially designed to be used on a range of elements.
  - They act on containers and provide means for various operations for the contents of the containers.
  - Algorithms:
    - Sorting
    - Searching
    - Important STL Algorithms
    - Useful Array algorithms
    - Partition Operations
- ✓ Iterators
  - Iterators are used to point at the memory addresses of STL containers.
- ✓ Functors
  - C++ functor (function object) is a class or struct object that can be called like a function.
  - Functors are used along with STLs

Vectors in C++ STL

- ✓ Vectors are the same as dynamic arrays with the ability to change themselves automatically when an element is inserted or deleted, with their storage being **handled automatically** by the container.
- ✓ Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- ✓ In vectors, data is inserted at the end.
- ✓ Initialize Vectors in C++
  - **vector <data-type> name (items)**
- ✓ Iterators
  - Helps to access the elements that are stored in a vector.
  - It’s an object that works like a pointer
    - **vector:: begin():** it gives an iterator that points to the first element of the vector.
    - **vector:: end():** it gives an iterator that points to the past-the-end element of the vector.
    - **vector::cbegin():** it’s the same as vector::begin(), but it doesn’t have the ability to modify elements.
    - **vector::cend():** it’s the same as vector::end() but can’t modify vector elements.
    - **vector:: rbegin():** Returns a reverse iterator pointing to the last element in the vector
      - It moves from last to first element
    - **vector:: rend():** Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

- ✓ Modifiers
  - used for changing the meaning of the specified data type
    - **vector::push_back():** This modifier pushes the elements from the back.
    - **vector::insert():** For inserting new items to a vector at a specified location.
    - **vector::pop_back():** This modifier removes the vector elements from the back.
    - **vector::erase():** It is used for removing a range of elements from the specified location.
    - **vector::clear():** It removes all the vector elements.
- ✓ Capacity
  - **Size()** –It returns the number of items in a vector.
  - **Max_size()** -It returns the highest number of items a vector can store.
  - **Capacity ()** –It returns the amount of storage space allocated to a vector.
  - **Resize ()** –It resizes the container to contain n items.
    - If the vector’s current size is greater than n, the back items will be removed from the vector.
    - If the vector’s current size is smaller than n, extra items will be added to the back of the vector.
  - **Empty ()** –it returns true if a vector is empty. Else, it returns false.

Example for Vectors

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
        vector<int> nums;
        for (int a = 1; a <= 5; a++)
                nums.push_back(a);
        cout << "Output from begin and end: ";
        for (auto a = nums.begin(); a != nums.end(); ++a)
                cout << *a << " ";
        cout << "\nOutput from rbegin and rend: ";
        for (auto a = nums.rbegin(); a != nums.rend(); ++a)
                cout << *a << " ";
    vector<int> v { 1, 2, 3, 4, 5 };
    cout << "\nOutput ";
    for (int x : v) {
          cout << x << " ";
    }
        return 0;
}
```

Sorting a vector in C++

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
         vector<int> v{ 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };

    sort(v.begin(), v.end());

    cout << "Sorted \n";
    for (auto x : v)
        cout << x << " ";

    return 0;
}
```

Deleting the elements from list

```
#include <algorithm>
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> lst{4, 1, 2, 3, 5};
           for(auto it = lst.begin(); it != lst.end();++it){
    if ((*it >3) ){
        it = lst.erase(it);
        --it;
           }
           }

           for(auto it:lst)cout<<it<<" ";
                      cout<<endl;
}
```

Problems on Vectors

- ✓ Write a C++ program that returns the elements in a vector that are strictly smaller than their adjacent left and right neighbours.
- ✓ Write a complete c++ program that uses 2 vectors, 1 for names (strings) and 1 for grades (longs).
  - Ask the user for the number of name/grade pairs that will be entered.
  - Get each of the names and grades.
  - Display the mean of the grades
  - Display the names of the students with their mean grade in descending order

List in C++ STL

- Lists are sequence containers that allow non-contiguous memory allocation.
- As compared to vector, the list has slow traversal,
  - but once a position has been found, insertion and deletion are quick
- List can be initialised in two ways.
  - list<int> new_list{1,2,3,4};
  - or
  - list<int> new_list = {1,2,3,4};

```
#include <algorithm>
#include <iostream>
#include <list>
using namespace std;
int main() {
        list<int> my_list = { 12, 5, 10, 9 };
        for (int x : my_list) {
            cout << x << '\n';
        }
}
```

C++ List Functions

Method	Description
insert()	It inserts the new element before the position pointed by the iterator.
push_back()	It adds a new element at the end of the vector.
push_front()	It adds a new element to the front.
pop_back()	It deletes the last element.
pop_front()	It deletes the first element.
empty()	It checks whether the list is empty or not.
size()	It finds the number of elements present in the list.
max_size()	It finds the maximum size of the list.
front()	It returns the first element of the list.
back()	It returns the last element of the list.
swap()	It swaps two list when the type of both the list are same.

Method	Description
reverse()	It reverses the elements of the list.
sort()	It sorts the elements of the list in an increasing order.
merge()	It merges the two sorted list.
splice()	It inserts a new list into the invoking list.
unique()	It removes all the duplicate elements from the list.
resize()	It changes the size of the list container.
assign()	It assigns a new element to the list container.
emplace()	It inserts a new element at a specified position.
emplace_back()	It inserts a new element at the end of the vector.
emplace_front()	It inserts a new element at the beginning of the list.

List Example

```
#include <iostream>
#include <list>
using namespace std;
int main(void) {
    list<int> l;
    list<int> l1 = { 10, 20, 30 };
    list<int> l2(l1.begin(), l1.end());
    list<int> l3(move(l1));
    cout << "Size of list l: " << l.size() << endl;
    cout << "List l2 contents: " << endl;
    for (auto it = l2.begin(); it != l2.end(); ++it)
        cout << *it << endl;
    cout << "List l3 contents: " << endl;
    for (auto it = l3.begin(); it != l3.end(); ++it)
              cout << *it << endl;
  }
```

Problems on Lists

- ✓ Write C++ program to perform following tasks on a list of integers
  - delete all odd numbers
  - Add the square of all existing elements of list
  - Find the sum of all elements of list.
- ✓ Given a set of integers. Perform the following tasks using C++ lists.
  - Arrange them in sorted order.
  - Input an integer and find the location of that if it is present.
  - if it is not present, find the index at which the smallest integer that is just greater than the given number is present.