1 Introduction to Programming

Thought process to solve a problem :

To solve a problem, what do we usually do?

1. The first thing is to understand the question.
2. Next is to see what values we are already given.
3. Next we think of an approach on how we could use the inputs given into the problem to bring out the solution. This approach can formally be expressed in two forms- We can use diagrams (aka flowcharts) to define the solution, or we can use pseudocode to define our solution.
4. Finally, if we have a solution, we code it using a Programming language.

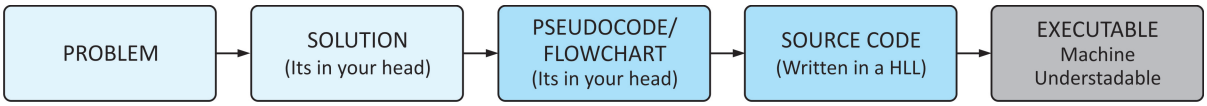


Figure 1 Steps to write a computer program

Using a Computer to solve a problem :

So, how can a computer solve a problem? As far as many of us know, it is a machine that can do some repetitive calculations. Ok, so lets' take this problem of finding whether the number 61 is prime or not?

Our approach could be like…

Is 61 divided by 2? No

Is 61 divided by 3? No

.
.
.

Is 61 divided by 2? No

Now, none of the numbers except 1 and 61 seems to completely divide 61. So, 61 is a prime number. Good! But, there is a lot of repetitive task going on. The task of dividing 61 with another number. For us, its tedious. Why not engage the computer into it?

So, we will **tell** the computer to go on dividing 61 by the numbers between 1 and 61. If any of the number divides, we say that 61 isn't prime. If none of them divides, then 61 is prime. So, this way we found a legitimate solution to the given problem.

But, an approach in natural language could be vague. For example consider I say, numbers between 1 and 61.

(a) It could be misunderstood by as the numbers including 1 and 61.

(b) Same problem in case of excluding 1 and including 61 (or) including 1 and excluding 61.

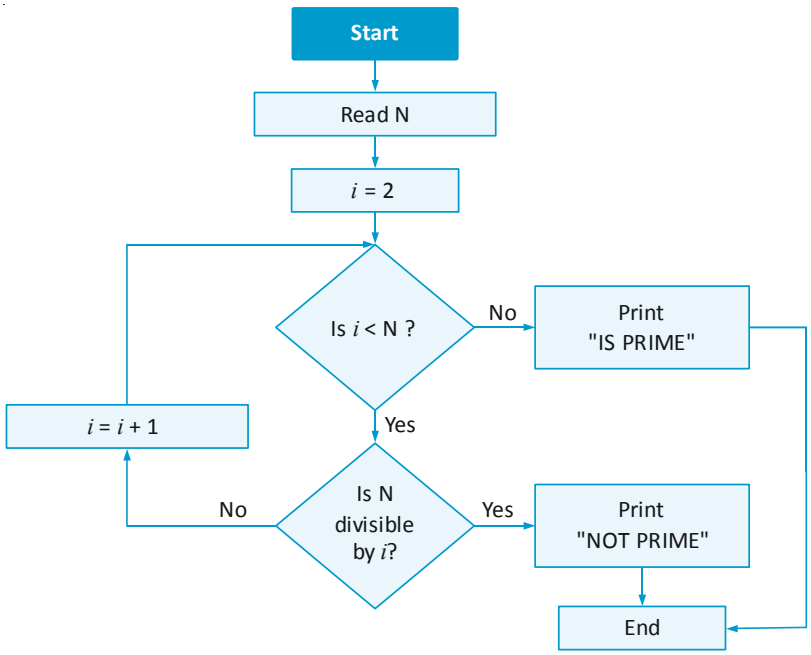
(c) Again, we need to tell what is meant by go on dividing by numbers.

So, natural language may lead to ambiguity and hence we need better ways to present out approach. To express thoughts unambiguously, flowcharts and pseudocodes are used.

Flowcharts :

A flowchart is a diagrammatic representation of an approach, where each step is put into a shape element (box, circle, etc.) and these elements are connected in the order of their execution.

Let's look back at the same problem of finding if a number 'n' is a prime number or not? We have understood that the approach would be to divide all numbers between 1 and n, by n. Now, let us draw a flowchart for it :



Pseudocode :

It resembles programming language, but is more generic in syntax. Its advantage is that programmers who code in different languages can explain their approach using pseudocode. Let's see some pseudocodes and their syntax:

Examples of Pseudocode :

1. Find Circumference and Area of a circle.

(i) Read R

(ii) PI = 3.14

(iii) Circumference = 2*PI*R

(iv) Area = PI*R*R

(v) Print Circumference

(vi) Print Area

2. Write a program to add N numbers taken from user.

(i) Sum = 0

(ii) I = 1

(iii) Read N

(iv) Until I <= N

- Read currentNumber
- Sum = Sum + currentNumber

(v) End while

(vi) Print Sum

(vii) Exit

3. Check if a given number N is prime.

(i) Read N

(ii) div = 2

(iii) While div < N do

- If N is divisible by div then
  - Print "NOT PRIME"
  - Exit
- End if
- div = div + 1

(iv) End while

(v) Print "IS PRIME"

(vi) Exit

4. Give grade to a student based on his/her marks. Above or at 90, give 'A', above or at 80 give 'B', above or at 60 give 'C' and above or at 40 give 'D'. Below 40 give 'F'.

```
Read N
If N >=90 then
        Print "A"
```

```
Else If N>=80 then
        Print “B”
Else If N>=60 then
        Print “C”
Else If N>=40 then
        Print “D”
Else then
        Print “F”
End If
Exit
```

In the above pseudocode, step number has been omitted. We can skip that in case indentation makes things clear.

1. Write a pseudocode to generate this pattern given N.

   For N=5

   ```
       1
      232
     34543
    4567654
   567898765
   ```

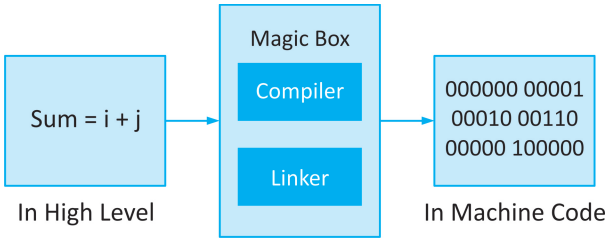
```
Read nLines
curRow = 1
While curRow <= nLines do
        nSpace = nLines – curRow
                curSpace = 1
                While curSpace <= nSpace
                        Print ‘ ‘
                 Val = curRow
        While  val < 2*curRow do
                Print val
                Val  = val + 1
        End while
        val = (2*curRow) – 2
        While val >= curRow
                Print val
                val = val – 1
        End while
        Print “\n”
End While
Exit
```

Now, when we are sure about our approach and we have expressed it unambiguously using flowchart/pseudocode, it is time to tell it to the machine. We code it using a programming language. This code is then converted into an executable and could be run on machine without the help of source code.

What is a programming language and why we need it :

A language using which, we can instruct the computer to carry out real life tasks and computations is called a programming language. It acts as a language in which we could easily express our thoughts to the machine.

Like natural languages, programming language has a fixed set of rules (called grammar or semantics) according to which programs (source code) could be written in it. These programs are then converted into a language which machines can understand (we call it binary language). This task is carried out by a special software (called compiler). Every language has its own complier/interpreter. So for C++ we are going to use g++ (included in GCC Compiler Suite), for java we'll use OpenJDK. Once a program is compiled and linked, its executable is created and the computer can run our program.



C++ compliation process

2 Getting Starting with C++ Programming

In the last chapter, we saw how to write pseudocodes that are more general and express our thoughts (solution to the problem) using a strict English-like language. In this chapter, we are going to learn, how to convert a pseudocode into C++ source code that can be compiled and run.

The Classical Hello World :

Now let us start our C++ programming by writing a simple hello world program and understand its syntax

```
1: # include <iostream>
2: using namespace std;
3:
4: int main(){
5:      //print Hello World
6:      cout << "Hello World!";
7:      //print a new line
8:      cout << endl;
9: }
```

We'll start reading the C++ code in the following sequence.

Line 4 : int main() : This is the line that is executed first whenever an executable is run. The '{' after main() defines a block which will be read (translated) sequentially.

Line 5 : It is called a comment. It starts with '//'. Any code (as of now) written after '//' is not meant to be translated by compiler. It serves as a reference for humans making code more readable.

Line 6 : It says that hey computer, just print Hello world on to the console (or screen).

Line 7 : It's again a comment

Line 8 : It is used to print an empty line (see it as hitting an enter on your keyboard after printing Hello World)

Line 9 : The '}' marks the end of the region started in line 4.

This nearly completes the code. However, C++ Compiler doesn't understand printing operation. So we tell it to import everything that there in 'iostream' (think it as a dictionary) where the operation printing is define. To achieve this operation **Line 1** is written. After doing that, C++ knows that printing can be done using 'cout'. So if you use anything, that's not understandable by C++ compiler, you need to first define its meaning and then use it.

Line 2 : As of now, just write it after all #includes line.

We have learnt how to print on the console using *cout << Things to print.*
Now we'll learn how to take input from user.

Taking input from User :

```
1: # include <iostream>
2: using namespace std;
3:
4: int main(){
5:   int x;
6:   cout << "Enter value for x ";
7:   cin >> x;   //Read into x
8: }
```

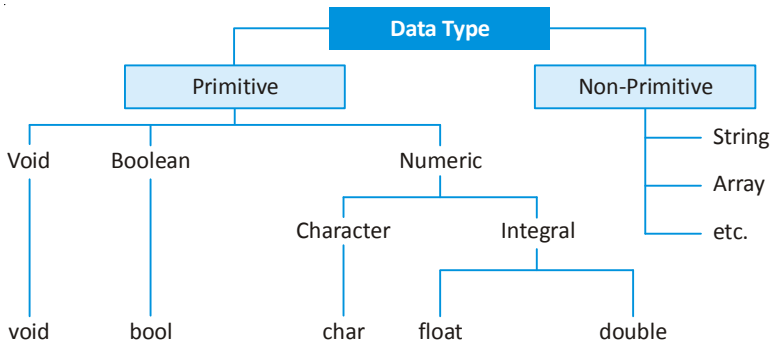
Line 5 : It tells compiler that x is a variable that will contain an integer. So we'll say the data type for x is int.

Line 7 : Read from keyboard(cin) to x.

Data Types in C++:

There are two types of data types:

- **Primitive data types**: These are basic data types which are used to represent single values. Primitive data types are predefined.
- **Non Primitive data types**: These are made up of Primitive data types. These are not defined by the C++ programming language, but are created by the user like arrays and strings. We will learn about them later!
- **Data type Modifiers:** These modifiers are used to increase or decrease the effective range of certain data types. They have lesser or greater memory usage depending on the increase/decrease in effective range. The 4 data type modifiers are - long, short, signed, unsigned.



Primitive Data Types : There are 6 types of primitive data types.

1. **int:** The int data type is a 32-bit (4 bytes) integer having values from -2^{31} to + ($2^{31} - 1$).
5. **char:** The char data type is a single 8-bit ASCII character having values from 0 to $2^8 - 1$.
6. **float:** The float data type is a single-precision 32-bit (4 bytes) floating point.
7. **double:** The double data type is a double-precision 64-bit (8 bytes) floating point.
8. **bool:** The bool data type can only contain two values true and false. Its size is 1 byte.
6. **void:** The void data type shows that it cannot contain anything. A variable cannot be defined for void type. However, a function which is declared of the void type doesn't return anything.

The following code, gives an insight to most commonly used data types for this course.

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

int main(){
    int num = 10;        //num stores an integer
    char alpha = 'a';    //alpha stores a character. Note the quotes.
    float pi = 3.14;
    //pi is a floating point variable that stores value 3.14
    double h = 6.626e-34;
    //floating point numbers can also be written in scientific notation
    //6.626e-34 = 6.626 * 10^-34
    bool isPrime = true;
    //isPrime is a var CAN store just 2 values either 0 or 1
}
```

Decision Making : Decision making is the integral part of most of the algorithms. This is achieved by the use of the 'If-else' statements!

We will understand it using an example. Consider writing a program to find largest of 3 numbers.

```
//Largest of 3 Numbers
# include <iostream>
using namespace std;

int main() {
    int no1;
    int no2;
    int no3;

    //Read 3 numbers
    cin >> no1 >> no2 >> no3;    //chaining or cascading
        /*
         Same as doing
```

```
        cin >> no1;
        cin >> no2;
        cin >> no3;
       */

    if (no1 > no2 and no1 > no3) {
        cout << no1 << " is largest" << endl;
    }
    else if (no2 > no1 && no2 > no1) {
    //same as if (no2 > no1 and no2 > no1)
        cout << no2 << " is largest" << endl;
    }
    else {
        cout << no3 << " is largest" << endl;
    }

}
```

Here the condition in the parenthesis of 'if' is evaluated. If the condition comes out to be true then the code in the '**if**' block is executed but if it comes out as false then the '**else**' block code is executed. If more than two conditions are there which needs to be checked then the *else if* ladder is used and hence multiple condition can be evaluated.

These conditions can also be nested which means that we can have an 'if-else' statements inside an 'if' block.

```
// Largest of 3 Numbers using nested if...else
# include <iostream>
using namespace std;

int main(){
    int no1, no2, no3;  //3 variables in one row
    if (no1 > no2){
        if (no1 > no3){
            cout << no1 << " is Largest" << endl;
        }
        else {
            cout << no3 << " is Largest" << endl;
        }
    }
    else {
        //this means no2 is larger than no1
        if (no2 > no3){
            cout << no2 << " is Largest" << endl;
        }
```

```
        else {
                cout << no3 << " is Largest" << endl;
            }
        }
 }
```

Looping : Most often while writing code, we need to perform a task over and over again till our work is done! This can be achieved by the concept of loops. In loops, we check a condition of 'Is the work done?' every time before doing the work. Thus, eventually when the work gets finished we break out of that loop. Note that it is very important to have a breaking condition in the loop else the loop would continue forever and it will become an infinite loop!

Let us try to convert 2nd pseudocode into C++ code. The task is to add N numbers where N will be given by user.

```
//Add N numbers input by user
# include <iostream>
using namespace std;
int main(){
      int N;
      cin >> N;

      int sum = 0;
      int x;
      int i = 1;           //number to be read

      while (i <= N){      //Loop begins here.
      //equivalent to UNTIL (i <= N)
          cin >> x;        //read x
          sum = sum + x;   //add x to sum
          i = i + 1;       //one number has been read. So read the next number
      }                    //Loop ends here.
      cout << sum << endl;
}
```

3 Arrays

Array Declaration and Usage

Suppose that you want to find the average of the marks for a class of 30 students, you certainly do not want to create 30 variables: mark1, mark2, ..., mark30. Instead, You could use a single variable, called an array, with 30 elements.

An array is a list of elements of the same type, identified by a pair of square brackets [ ]. To use an array, you need to declare the array with 3 things: a name, a type and a dimension (or size, or length). The syntax is:

```
 type arrayName[arraylength];  // arraylength can only be a literal/constant

// I recommend using a plural name for array, e.g., marks, rows, numbers.
int marks[5];         // Declare an int array called marks with 5 elements
double numbers[10];   // Declare an double array of 10 elements
const int SIZE = 9;
float temps[SIZE];    // Use const int as array length
// Some compilers support an variable as array length, e.g.,
int size;
cout << "Enter the length of the array: ";
cin >> size;
float  values[size];
```

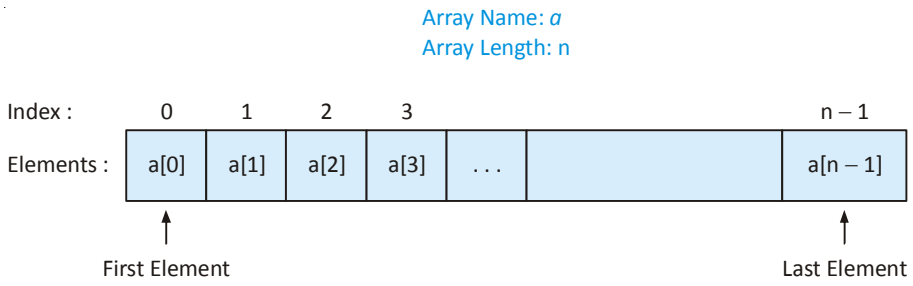
Take note that, in C++, the value of the elements are undefined after declaration.
You can also initialize the array during declaration with a comma-separated list of values, as follows:

```
// Declare and initialize an int array of 3 elements
int numbers[3] = {11, 33, 44};
// If length is omitted, the compiler counts the elements
int numbers[] = {11, 33, 44};
// Number of elements in the initialization shall be equal to or less than
length
int numbers[5] = {11, 33, 44};
// Remaining elements are zero. Confusing! Don't do this
```

```
int numbers[2] = {11, 33, 44};  // ERROR: too many initializers
// Use {0} or {} to initialize all elements to 0
int numbers[5] = {0};  // First element to 0, the rest also to zero
int numbers[5] = {};   // All element to 0 too
/* Test local array initialization (TestArrayInit.cpp) */
#include <iostream>
using namespace std;
int main() {
   int const SIZE = 5;
   int a1[SIZE];   // Uninitialized
   for (int i = 0; i < SIZE; ++i) cout << a1[i] << " ";
   cout << endl;   // ? ? ? ? ?

   int a2[SIZE] = {21, 22, 23, 24, 25}; // All elements initialized
   for (int i = 0; i < SIZE; ++i) cout << a2[i] << " ";
   cout << endl;   // 21 22 23 24 25

   int a3[] = {31, 32, 33, 34, 35};   // Size deduced from init values
    int a3Size = sizeof(a3)/sizeof(int);
   cout << "Size is " << a3Size << endl;   // 5
   for (int i = 0; i < a3Size; ++i) cout << a3[i] << " ";
   cout << endl;   // 31 32 33 34 35

   int a4[SIZE] = {41, 42};  // Leading elements initialized, the rests to 0
   for (int i = 0; i < SIZE; ++i) cout << a4[i] << " ";
   cout << endl;   // 41 42 0 0 0

   int a5[SIZE] = {0};  // First elements to 0, the rests to 0 too
   for (int i = 0; i < SIZE; ++i) cout << a5[i] << " ";
   cout << endl;   // 0 0 0 0 0

   int a6[SIZE] = {};   // All elements to 0 too
   for (int i = 0; i < SIZE; ++i) cout << a6[i] << " ";
   cout << endl;   // 0 0 0 0 0
}
```

You can refer to an element of an array via an index (or subscript) enclosed within the square bracket [ ]. C++'s array index begins with zero. For example, suppose that marks is an int array of 5 elements, then the 5 elements are: marks[0], marks[1], marks[2], marks[3], and marks[4].

```
// Declare & allocate a 5-element int array
int marks[5];
// Assign values to the elements
marks[0] = 95;
marks[1] = 85;
marks[2] = 77;
marks[3] = 69;
marks[4] = 66;
cout << marks[0] << endl;
cout << marks[3] + marks[4] << endl;
```



To create an array, you need to known the length (or size) of the array in advance, and allocate accordingly. Once an array is created, its length is fixed and cannot be changed. At times, it is hard to ascertain the length of an array (e.g., how many students in a class?). Nonetheless, you need to estimate the length and allocate an upper bound. This is probably the major drawback of using an array. C++ has a vector template class (and C++11 added an arraytemplate class), which supports dynamic resizable array.

You can find the array length using expression sizeof(arrayName)/sizeof(arrayName[0]), where sizeof(arrayName) returns the total bytes of the array and sizeof(arrayName[0]) returns the bytes of first element.

C/C++ does not perform array index-bound check. In other words, if the index is beyond the array's bounds, it does not issue a warning/error. For example,

```
const int size = 5;
int numbers[size];  // array index from 0 to 4
// Index out of bound!
// Can compiled and run, but could pose very serious side effect!
numbers[88] = 999;
cout << numbers[77] << endl;
```

This is another pitfall of C/C++. Checking the index bound consumes computation power and depicts the performance. However, it is better to be safe than fast. Newer programming languages such as Java/C# performs array index bound check.

Array and Loop

Arrays works hand-in-hand with loops. You can process all the elements of an array via a loop, for example,

```
/*
 *   Find the mean and standard deviation of numbers kept in an array
(MeanStdArray.cpp).
 */
#include <iostream>
#include <iomanip>
#include <cmath>
#define SIZE 7
using namespace std;

int main() {
   int marks[] = {74, 43, 58, 60, 90, 64, 70};
   int sum = 0;
   int sumSq = 0;
   double mean, stdDev;
   for (int i = 0; i < SIZE; ++i) {
      sum += marks[i];
       sumSq += marks[i]*marks[i];
   }
    mean = (double)sum/SIZE;
    cout << fixed << "Mean is " << setprecision(2) << mean << endl;

    stdDev = sqrt((double)sumSq/SIZE - mean*mean);
    cout << fixed << "Std dev is " << setprecision(2) << stdDev << endl;

   return 0;
}
```

Range-based for loop (C++11) :

C++11 introduces a range-based for loop (or for-each loop) to iterate through an array, as illustrated in the following example:

```
/* Testing For-each loop (TestForEach.cpp) */
#include <iostream>
using namespace std;
int main() {
   int numbers[] = {11, 22, 33, 44, 55};
   // For each member called number of array numbers - read only
   for (int number : numbers) {
      cout << number << endl;
   }
   // To modify members, need to use reference (&)
   for (int &number : numbers) {
      number = 99;
   }
   for (int number : numbers) {
      cout << number << endl;
   }
   return 0;
}
```

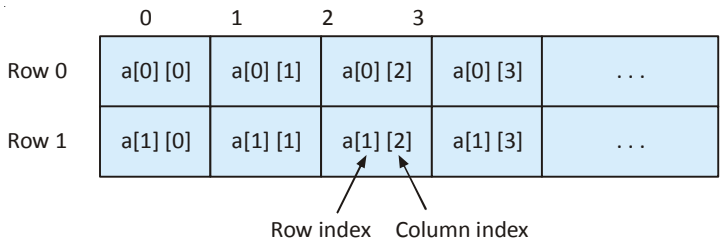
To compile the program under GNU GCC (g++), you may need to specify option -std=c++0x or -std=c++11:

```
g++ -std=c++0x -o TestForEach.exe TestForEach.cpp
// or
g++ -std=c++11 -o TestForEach.exe TestForEach.cpp
```

Multi-Dimensional Array

For example

int mat[2][3] = { {11, 22, 33}, {44, 55, 66} };



For 2D array (table), the first index is the row number, second index is the column number. The elements are stored in a so-called row-major manner, where the column index runs out first.

Example :

```
/* Test Multi-dimensional Array (Test2DArray.cpp) */
#include <iostream>
using namespace std;
void printArray(const int[][3], int);

int main() {
   int myArray[][3] = {{8, 2, 4}, {7, 5, 2}}; // 2x3 initialized
                  // Only the first index can be omitted and implied
   printArray(myArray, 2);
   return 0;
}
// Print the contents of rows-by-3 array (columns is fixed)
// Functions coming soon!
void printArray(const int array[][3], int rows) {
   for (int i = 0; i < rows; ++i) {
      for (int j = 0; j < 3; ++j) {
         cout << array[i][j] << " ";
      }
      cout << endl;
   }
}
```

Array of Characters - C-String

In C, a string is a char array terminated by a NULL character '\0' (ASCII code of Hex 0). C++ provides a new string class under header <string>. The original string in C is known as C-String (or C-style String or Character String). You could allocate a C-string via:

```
char message[256];    // Declare a char array
// Can hold a C-String of up to 255 characters terminated by '\0'
char str1[] = "Hello"; // Declare and initialize with a "string literal".
// The length of array is number of characters + 1 (for '\0').
char str1char[] = {'H', 'e', 'l', 'l', 'o', '\0'};  // Same as above
char str2[256] = "Hello";
// Length of array is 256, keeping a smaller string of length 5+1=6.
```

Example **:**

You can use cin and cout to handle C-strings.

cin << reads a string delimited by whitespace;

cin.getline(var, size) reads a string of into var till newline of length up to size-1, discarding the newline (replaced by '\0'). The size typically corresponds to the length of the C-string array.

cin.get(var, size) reads a string till newline, but leaves the newline in the input buffer.

cin.get(), without argument, reads the next character.

```
/* Test C-string (TestCString.cpp) */
#include <iostream>
using namespace std;

int main() {
   char msg[256]; // Hold a string of up to 255 characters (terminated by '\0')

   cout << "Enter a message (with space)" << endl;
   cin.getline(msg, 256);  // Read up to 255 characters into msg
   cout << msg << endl;

   // Access via null-terminated character array
   for (int i = 0; msg[i] != '\0'; ++i) {
      cout << msg[i];
   }
   cout << endl;

   cout << "Enter a word (without space)" << endl;
   cin >> msg;
   cout << msg << endl;

   // Access via null-terminated character array
   for (int i = 0; msg[i] != '\0'; ++i) {
      cout << msg[i];
   }
   cout << endl;
   return 0;
}
```

4 Functions

Say, we want to compute factorial of two different numbers. For that the program would be :

```
01: //factorial of a number
02:
03: # include <iostream>
04: using namespace std;
05:
06: int main(){
07:     int n1, n2;
08:     cin >> n1 >> n2;
09:
10:     int fact1 = 1;
11:     int fact2 = 1;
12:
13:     //computing factorial of n1
14:     for(int i = 2; i <= n1; ++i) fact1 = fact1 * i;
15:
16:     //computing factorial of n2
17:     for(int i = 2; i <= n2; ++i) fact2 = fact2 * i;
18:
19:     cout << "Factorial of " << n1 << " " << fact1 << endl;
20:     cout << "Factorial of " << n2 << " " << fact2 << endl;
21: }
```

The above program has the following problems :

(a) Repetition of Code : Code in line 14 and 17 performs a task that is logically same i.e. to compute factorial of a number.

(b) Non-maintainable : If there is some error while developing the solution for factorial and later it needs to be corrected, then all the lines where factorial is computed need to be updated.

Let us look at the following program that is written using functions.

```
01: //factorial using function
02:
03: # include <iostream>
```

```
04: using namespace std;
05:
06: int factorial(int n);    //function declaration
07:
08: int main(){
09:     int n1, n2;
10:     cin >> n1 >> n2;
11:
12:      int fact1 = factorial(n1);   //function invocation
13:      int fact2 = factorial(n2);   //or simply function calling
14:
15:     cout << "Factorial of " << n1 << " " << fact1 << endl;
16:     cout << "Factorial of " << n2 << " " << fact2 << endl;
17: }
18:
19: //function definition
20: int factorial(int n){
21:     int fact = 1;
22:
23:     //computing factorial of n
24:     for(int i = 2; i <= n; ++i) fact = fact * i;
25:
26:     return fact;
27: }
```

Here the code for computing factorial has been written only once [20, 27][1] but used twice [12], [13]. [6] is a function declaration that provides the following 4 information about the function.

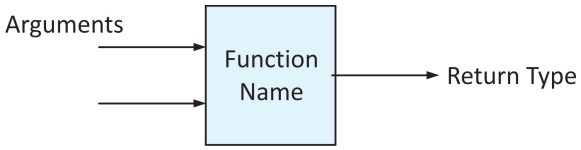
(a) Function name : factorial

(b) Return type : int

(c) Number of arguments : 1

(d) Function arguments : a variable of int type

Note that it does NOT tell us anything about what the function does. It just tell us how it looks.



Function definition starts from [20]. It tells the compiler what the function does and how it does so. Had the function definition done before function calling [12], the function declaration becomes optional.

[1] [i] represents i^{th} line number

Now we can define function formally. A **function** is a set of code which is referred to by a name and can be called (invoked) at any point in the program by using the function's name. It is like a subprogram where we can pass parameters to it and it can even return a value.

By the return type we mean that we specify the data type of the return value. It may also happen sometimes that the function does not return anything. In that case the return type of the function should be **'void'**.

- If no return type is mentioned, by default int is assumed.
- If no return statement exists in a function which is expected to return some value, some garbage value is returned.
- You cannot return a value from a function with void return type.

Function Calling :

Now just making the function would not execute it even when the program is run. To execute any function we need to invoke it. This is called **calling the function**. Function can be called from the main function and also from other functions. The called function must either be declared or be defined before the calling function. Otherwise, the calling function cannot find the called function, and it will lead to a compile time error. If there are any parameters to be given to the function, they are also passed within the parenthesis. If the function returns any value it can be stored in a variable. Now let us write two functions and see how they can be called!

When the function is called, we can pass the parameters of the same data type as specified in its definition. Note that while calling the function we only pass in the names of the variables or values and don't specify its data type!

```
01: //sum function
02: # include <iostream>
03: using namespace std;
04:
05: int add(int num1, int b){
06:     return num1 + b;
07: }
08:
09: int main(){
10:     int num1 = 3;
11:     int num2 = 5;
12:     int sum = add(num1, num2);
13:     cout << sum << endl;
14:     return 0;
15: }
```

In the above code, we have written a function for adding two numbers. An important point to note here is that the variables present in the main function are different from those that are present in the add function, though they might have same names. To understand this better let us learn about the concept of variables and their scope.

Variables and their Scope :

During runtime a memory pool is created by the program to store all the variables, codes for functions, etc. In this memory pool there are majorly two types of memory – the **stack memory** and the **heap memory**. Any variables that are statically declared (their size if known at compile time) are stored in the stack memory, whereas any variables/objects that are created dynamically at runtime (their size is not known at compile time), are created in the heap memory. All the functions that are called occupy their area on the stack where all their variables are stored. The functions keep stacking one above the other in the order of their calls. When the function returns, it is erased from the stack and all its local variables are destroyed.

The variables that are present in one function are local to only that function. It cannot be accessed outside that function. For instance a variable made in the main function is not available to the add function unless we pass it as a parameter to it. However, using pass by reference, a variable defined in one function can be accessed in other function.

Consider the following example of swapping to numbers using function.

```
01: //call by reference
02: # include <iostream>
03: using namespace std;
04:
05: void swp1(int x, int y){
06:     int tmp = x;
07:     x = y;
08:     y = tmp;
09: }
10:
11: void swp2(int& x, int& y){
12: //x and y are references...call by reference
13:     int tmp = x;
14:     x = y;
15:     y = tmp;
16: }
17:
18: int main() {
19:     int a, b;
```

```
20:      cin >> a >> b;
21:
22:      swp1(a, b);
23:      cout << a << " " << b << endl;   //No swapping
24:
25:      swp2(a, b);
26:      cout << a <<  " " << b << endl; //values actually swapped
27: }
```

Note the difference in line 5 and line 11. In line 11, & is used to refer to an already existing VARAIBLE. Also, there is no change in the calling of function.

Difference between Pass by Value v/s Pass by Reference:

PASS BY VALUE	PASS BY REFERENCE
Changes in the formal parameters are NOT reflected back to the actual parameters (arguments).	Changes in the formal parameters are reflected to the actual parameters.
New variables are created.	Already exiting variables are given new names.

5 Recursion

Recursion is a technique (just like loops) where the solution to a problem depends on the solution to smaller instances of that problem. In cases where we need to solve a bigger problem, we try to reduce that bigger problem in a number of smaller problems and then solve them. For instance we need to calculate 2^8. We can do this by first calculating 2^4 and then multiplying 2^4 with 2^4. Now to calculate 2^4, we can calculate 2^2 which in turn depend on just 2^1. So for calculating 2^8, we just need to calculate 2^1 and then keep on multiplying the obtained numbers until we reach the result.

So by dividing the problem in smaller problems we can solve the original program easily. But we can't go on making our problem smaller infinitely. We need to stop somewhere and terminate. This point is called a **base case**. The base case tells us that the problem doesn't have to be divided further. For instance, in the above case when power of 2 equals to 1 we can return 2. So power being equal to 1 becomes our base case.

```
int power(int x,int n)
{
      if(n==1)
      {
        return x;
      }
      int  smallPow=power(x,n/2);
      if(n%2==0)  {
            return  smallPow*smallPow;
      }
      else {
            return  smallPow*smallPow*x;
      }
}
```

In the next program we are calculating the factorial of a number.

For calculating the factorial of n, all we need is the factorial for n-1 and then multiply it with the number itself. As we can see that factorial of a bigger number depends on the factorial of a smaller number, thus this problem can be solved by recursion.

So for calculation of factorial of a bigger number, we first calculate factorial of n-1 and then multiply it by n. If the number is 1, we don't need to calculate anything but just return 1. Thus this is our base case.

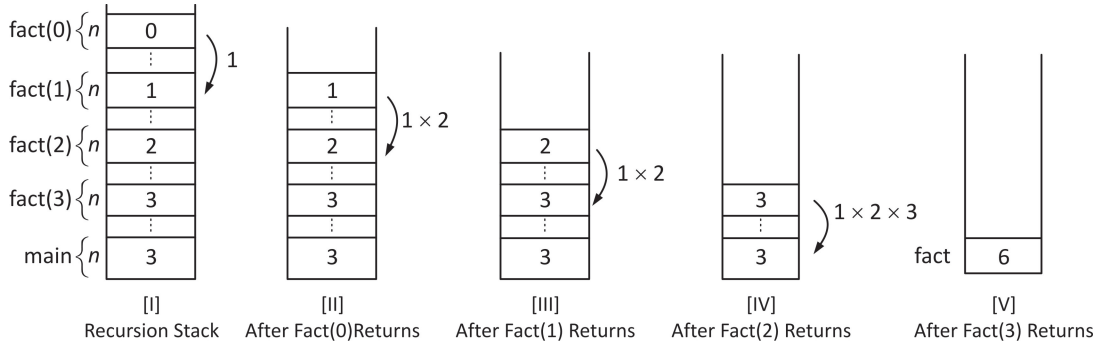
Notice that in the above code the factorial function calls itself on a smaller input and then use that result to calculate the final answer. Thus in recursion, a function calls itself.

An important point to note here is that in the base case we need to have a return statement compulsorily even if the return type of the function is void. This is because even in the base case if we don't have the return statement, the code after the base case would begin to execute and the recursion would execute infinitely. Thus to terminate recursion, we need to have a return statement in the base case.

For factorial of 5:

```
factorial(5)
    factorial(4)
        factorial(3)
            factorial(2)
                factorial(1)
                    return 1
                return 2*1 = 2
            return 3*2 = 6
        return 4*6 = 24
    return 5*24 = 120
```

Let us understand the recursive function calls through memory maps:



Now we know that every recursive function includes **a base case and a call to itself**. So every time the function calls itself we have two opportunities to do our work. First is before calling itself and second is after the call. First opportunity helps to do work on the input whereas second opportunity lets us to do work on the result for the smaller problem. To understand this better let us write code for printing a series of increasing and decreasing numbers.

```
void printID(int n)
{
  if(n==0)
       return;

   cout<<n<<endl;
   printID(n-1);
   cout<<n<<endl;
}
```

Every time a recursive function is called, the called function is pushed into the running program stack over the calling function. This uses up memory equal to the data members of a function. So, using recursion is recommended only upto certain depth. Otherwise too much memory would be wasted in stack space for program.

Limitation of Recursive Call :

There is an upper limit to the number of recursive calls that can be made. To prevent this make sure that your base case is reached before stack size limit exceeds.
So, if we want to solve a problem using recursion, then we need to make sure that:

- The problem can broken down into smaller problems of same type.
- Problem has some base case(s).
- Base case is reached before the stack size limit exceeds.

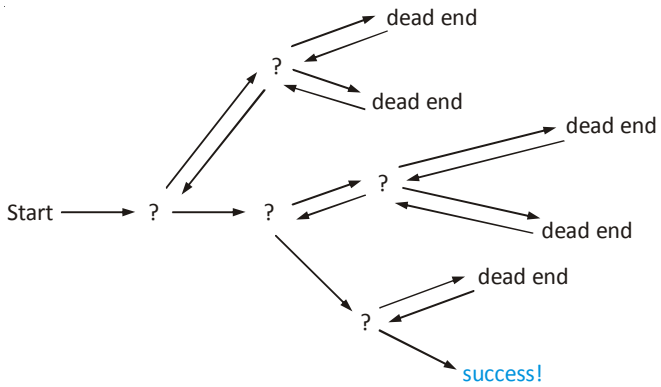
Backtracking :

When we solve a problem using recursion, we break the given problem into smaller ones. Let's say we have a problem **PROB** and we divided it into three smaller problems **P1, P2** and **P3**. Now it may be the case that the solution to PROB does not depend on all the three subproblems, in fact we don't even know on which one it depends.

Let's take a situation. Suppose you are standing in front of three tunnels, one of which is having a bag of gold at its end, but you don't know which one. So you'll try all three. First go in tunnel 1, if that is not the one, then come out of it, and go into tunnel 2, and again if that is not the one, come out of it and go into tunnel 3. So basically in backtracking we attempt solving a subproblem, and if we don't reach the desired solution, then undo whatever we did for solving that subproblem, and again try solving another subproblem.

Basically **Backtracking** is an algorithmic paradigm that tries different solutions until finds a solution that "works". Problems which are typically solved using backtracking technique have following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once.

Note : We can solve this by using **recursion method**.



Problems based on Backtracking :

1. **N-Queen Problem :** Given a chess board having N×N cells, we need to place N queens in such a way that no queen is attacked by any other queen. A queen can attack horizontally, vertically and diagonally.

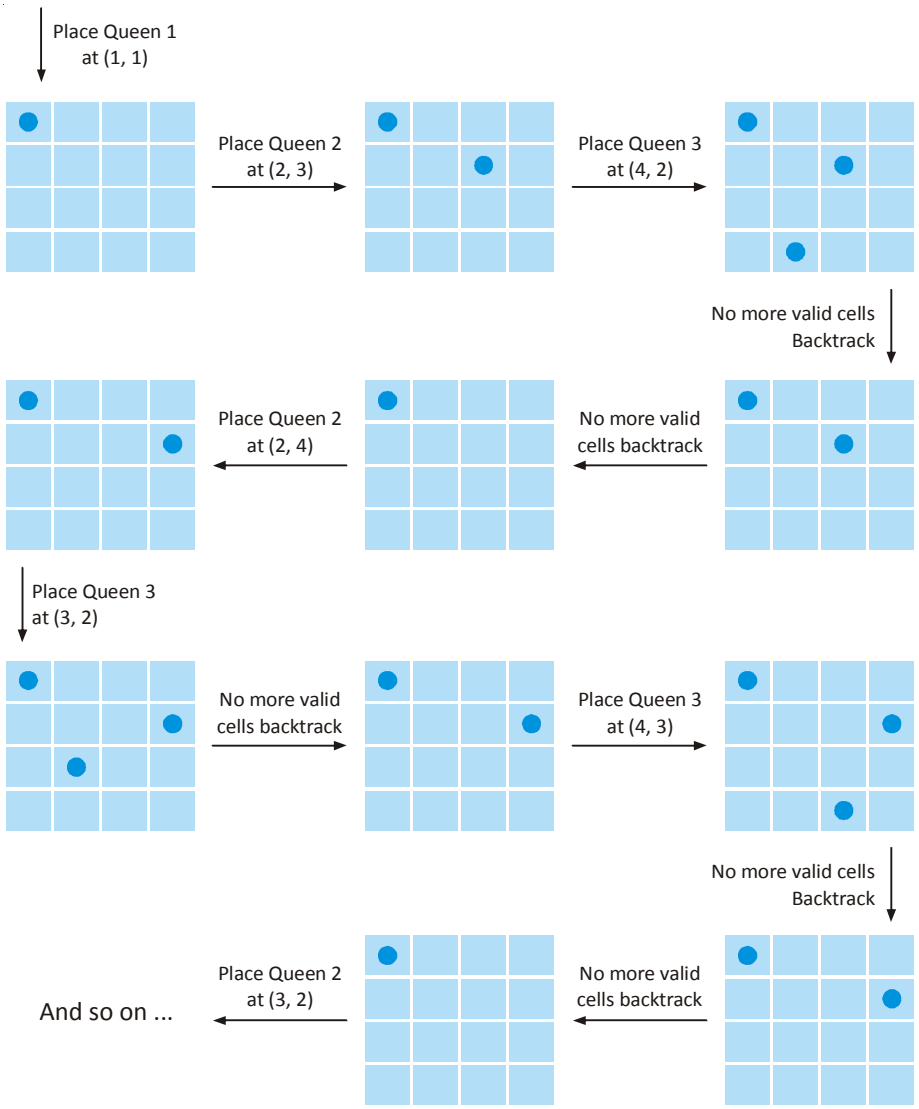
Approach towards the Solution:

We are having N × N unattacked cells where we need to place N- queens. Let's place the first queen at a cell (i, j), so now the number of unattacked cells is reduced, and number of queens to be placed is N – 1. Place the next queen at some unattacked cell. This again reduces the number of unattacked cells and number of queens to be placed becomes N – 2. Continue doing this, as long as following conditions hold.

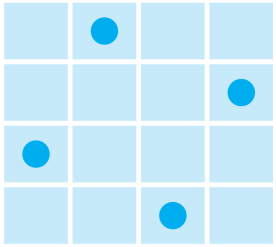
- The number of unattacked cells is not equal to 0.
- The number of queens to be placed is not equal to 0.

If the number of queens to be placed becomes 0, then it's over, we found a solution. But if the number of unattacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell.

Let's N = 4, We have to place 4 queen in 4×4 cells.



So, final solution of this problem is



So, clearly, we tries solving a subproblem, if that does not result in the solution, it undo whatever changes were made and solve the next subproblem. If the solution does not exists (like N=2), then it returns false.

4. Rat in a Maze :

Given a maze, NxN matrix. A rat has to find a path from source to des-ti-na-tion. maze[0][0] (left top corner)is the source and maze[N-1][N-1](right bot-tom cor-ner) is des-ti-na-tion. There are few cells which are blocked, means rat can-not enter into those cells. The rat can move only in two directions: forward and down.

Approach Towards the Solution :

Here we have to generate all paths from source to destination and one by one check if the generated path satisfies the constraints or not.

Let's the given maze is in the form of matrix whose entries are either 0 or 1. Entry 0 represent the block path from where the rat can't move. We have to print another matrix whose entries are either 0 or 1. In output matrix, entry 1 represents the the path of Rat from starting point to destination point.

Algorithm to generate all the Paths :

While there are untried paths

```
{
     Generate the next paths
     If this path has all blocks as 1
       {
              Print this path;
       }
}
```

Backtracking Algorithm :

If destination is reached
  Print the solution
Else
  {

a) Mark current cell in solution matrix as 1.
b) Move forward in horizontal direction and recursively check if this move leads to a solution.
c) If the move chosen in the above step doesn't lead to a solution then move down and check of this move leads to a solution.
d) If none of the above solutions work then unmark this cell as 0 (Backtrack) and return false.

  }

6 Pointers

Pointers, References and Dynamic Memory Allocation are the most powerful features in C/C++ language, which allows programmers to directly manipulate memory to *efficiently manage the memory* - the most critical and scarce resource in computer - *for best performance*. However, "pointer" is also the most complex and difficult feature in C/C++ language.

Pointers are extremely powerful because they allows you to access addresses and manipulate their contents. But they are also extremely complex to handle. Using them correctly, they could greatly improve the efficiency and performance. On the other hand, using them incorrectly could lead to many problems, from un-readable and un-maintainable codes, to infamous bugs such as memory leaks and buffer overflow, which may expose your system to hacking. Many new languages (such as Java and C#) remove pointer from their syntax to avoid the pitfalls of pointers, by providing automatic memory management.

Although you can write C/C++ programs without using pointers, however, it is difficult not to mention pointer in teaching C/C++ language. Pointer is probably not meant for novices and dummies.

Pointer Variables :

A *computer memory location* has an *address* and holds a *content*. The *address* is a numerical number (often expressed in hexadecimal), which is hard for programmers to use directly. Typically, each address location holds 8-bit (i.e., 1-byte) of data. It is entirely up to the programmer to interpret the meaning of the data, such as integer, real number, characters or strings.

To ease the burden of programming using numerical address and programmer-interpreted data, early programming languages (such as C) introduce the concept of variables. A variable is a *named* location that can store a *value* of a particular *type*. Instead of numerical addresses, names (or identifiers) are attached to certain addresses. Also, types (such as int, double, char) are associated with the contents for ease of interpretation of data.

Each address location typically hold 8-bit (i.e., 1-byte) of data. A 4-byte int value occupies 4 memory locations. A 32-bit system typically uses 32-bit addresses. To store a 32-bit address, 4 memory locations are required.

The following diagram illustrate the relationship between computers' memory *address* and *content*; and variable's *name*, *type* and *value* used by the programmers.

Pointer Variables (or Pointers) :

A *pointer variable* (or *pointer* in short) is basically the same as the other variables, which can store a piece of data. Unlike normal variable which stores a value (such as an int, a double, a char), a *pointer stores a memory address*.

Declaring Pointers :

Pointers must be declared before they can be used, just like a normal variable. The syntax of declaring a pointer is to place a * in front of the name. A pointer is associated with a type (such as int and double) too.

```
type *ptr; // Declare a pointer variable called ptr as a pointer of type
// or
type * ptr;
// or
type *ptr ;     // I shall adopt this convention
```

For Example,

```
int * iPtr;
// Declare a pointer variable called iPtr pointing to an int (an int pointer)
// It contains an address. That address holds an int value.
double * dPtr; // Declare a double pointer
```

Take note that you need to place a * in front of each pointer variable, in other words, * applies only to the name that followed. The * in the declaration statement is not an operator, but indicates that the name followed is a pointer variable. For example,

```
int *p1, *p2, i ;      // p1 and p2 are int pointers. i is an int
int *p1, p2, i ;       // p1 is a int pointer, p2 and i are int
int *p1, *p2, i ;      // p1 and p2 are int pointers, i is an int
```

Naming Convention of Pointers: Include a "p" or "ptr" as *prefix* or *suffix*,
e.g., iPtr, numberPtr, pNumber, pStudent.

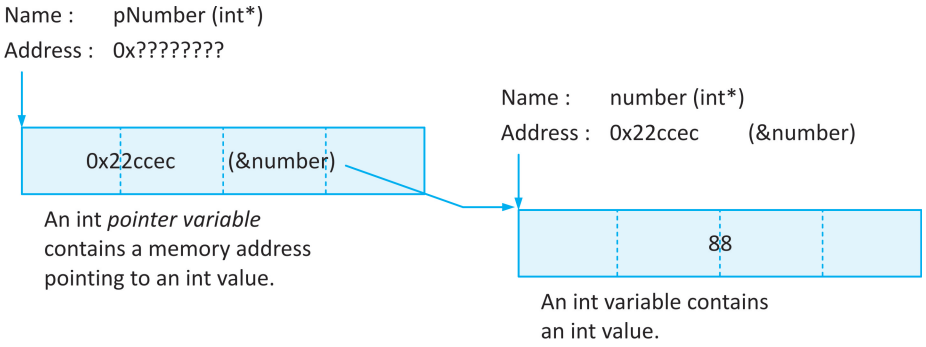
Initializing Pointers via the Address-Of Operator (&) :

When you declare a pointer variable, its content is not initialized. In other words, it contains an address of "somewhere", which is of course not a valid location. This is dangerous! You need to initialize a pointer by assigning it a valid address. This is normally done via the *address-of operator* (&).

The *address-of operator* (&) operates on a variable, and returns the address of the variable. For example, if number is an int variable, &number returns the address of the variable number.

You can use the address-of operator to get the address of a variable, and assign the address to a pointer variable. For example,

```
int number = 88 ;      // An int variable with a value
int * pNumber ;
// Declare a pointer variable called pNumber pointing to an int (or int pointer)
pNumber = &number;
// Assign the address of the variable number to pointer pNumber.
int * pAnther = &number;
// Declare another int pointer and init to address of the variable number
```



As illustrated, the int variable number, starting at address 0x22ccec, contains an int value 88. The expression &number returns the address of the variable number, which is 0x22ccec. This address is then assigned to the pointer variable pNumber, as its initial value.

The address-of operator (&) can only be used on the RHS.

Indirection or Dereferencing Operator (*) :

The *indirection operator* (or *dereferencing operator*) (*) operates on a pointer, and returns the value stored in the address kept in the pointer variable. For example, if pNumber is an int pointer, *pNumberreturns the int value *"pointed to"* by pNumber.

For Example,

```
int number 88;
int * pNumber = &number;
// Declare and assign the address of variable number to pointer pNumber (0x22ccec)
cout << pNumber << endl ;
// Print the content of the pointer variable, which contain an address (0x22ccec)
cout << *pNumber << endl;
// Print the value "pointed to" by the pointer, which is an int (88)
*pNumber = 99;
// Assign a value to where the pointer is pointed to, NOT to the pointer variable
```

```
cout << *pNumber << endl;
// Print the new value "pointed to" by the pointer (99)
cout << number << endl;
// The value of variable number changes as well (99)
```

Take note that pNumber stores a memory address location, whereas *pNumber refers to the value stored in the address kept in the pointer variable, or the value pointed to by the pointer.
As illustrated, a variable (such as number) *directly* references a value, whereas a pointer *indirectly* references a value through the memory address it stores. Referencing a value indirectly via a pointer is called *indirection* or *dereferencing*.

The indirection operator (*) can be used in both the RHS (temp = *pNumber) and the LHS (*pNumber = 99) of an assignment statement.

Take note that the symbol * has different meaning in a declaration statement and in an expression. When it is used in a declaration (e.g., int * pNumber), it denotes that the name followed is a pointer variable. Whereas when it is used in a expression (e.g., *pNumber = 99; temp << *pNumber;), it refers to the value pointed to by the pointer variable.

Pointer has a Type Too :

A pointer is associated with a type (of the value it points to), which is specified during declaration. A pointer can only hold an address of the declared type; it cannot hold an address of a different type.

```
int i = 88;
double d = 55.66;
int *iPtr = &i;        // int pointer pointing to an int value
double * dPtr = &d ;   // double pointer pointing to a double value
iPtr = &d ;      // ERROR, cannot hold address of different type
dPtr = &i ;      // ERROR
iPtr = i ;       // ERROR, pointer holds address of an int, NOT int value
int j = 99 ;
iPtr = &j ;      // You can change the address stored in a pointer
```

Uninitialized Pointers :

The following code fragment has a serious logical error!

```
int * iPtr;
*iPtr = 55;
cout << *iPtr << endl;
```

Null Pointers :

You can initialize a pointer to 0 or NULL, i.e., it points to nothing. It is called a *null pointer*. Dereferencing a null pointer (*p) causes an STATUS_ACCESS_VIOLATION exception.

```
int * iPtr = 0 ;
// Declare an int pointer, and initialize the pointer to point to nothing
cout << *iPtr << endl;
// ERROR! STATUS_ACCESS_VIOLATION exception
int * p = NULL ;       // Also declare a NULL pointer points to nothing
```

Initialize a pointer to null during declaration is a good software engineering practice.
C++11 introduces a new keyword called nullptr to represent null pointer.

Reference Variables

C++ added the so-called *reference variables* (or *references* in short). A reference is an *alias*, or an *alternate name* to an existing variable. For example, suppose you make peter a reference (alias) to paul, you can refer to the person as either peter or paul.
The main use of references is acting as function formal parameters to support pass-by-reference. In an reference variable is passed into a function, the function works on the original copy (instead of a clone copy in pass-by-value). Changes inside the function are reflected outside the function.
A reference is similar to a pointer. In many cases, a reference can be used as an alternative to pointer, in particular, for the function parameter.

References (or Aliases) (&) :

Recall that C/C++ use & to denote the *address-of* operator in an expression. C++ assigns an additional meaning to & in declaration to declare a reference variable.

The meaning of symbol & is different in an expression and in a declaration. When it is used in an expression, & denotes the address-of operator, which returns the address of a variable, e.g., if number is an intvariable, &number returns the address of the variable number (this has been described in the above section).

Howeve, when & is used in a *declaration* (including *function formal parameters*), it is part of the type identifier and is used to declare a *reference variable* (or *reference* or *alias* or *alternate name*). It is used to provide *another name*, or *another reference*, or *alias* to an existing variable.

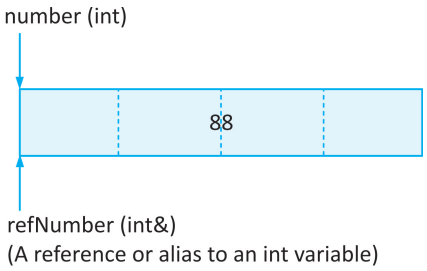
The syntax is as follow:

```
type &newName = existingName ;
// or
type&newName = existingName ;
// or
type&newName = existingName ;     // I shall adopt this convention
```

It shall be read as "*newName is a reference to exisitngName*", or "*newNew is an alias of existingName*". You can now refer to the variable as *newName* or *existingName*.

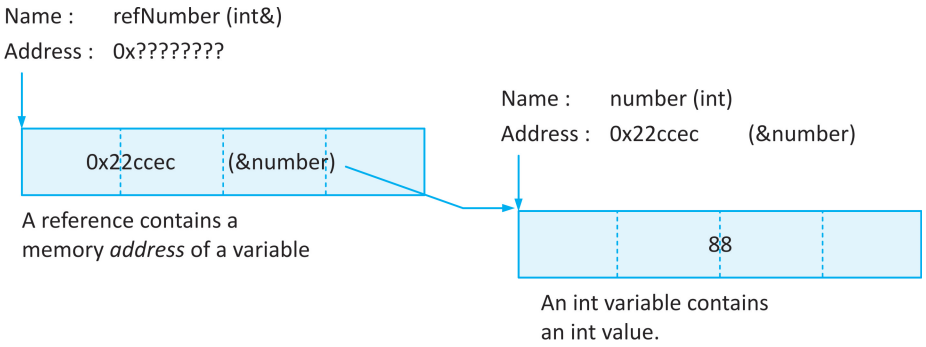
For example,

```
/* Test reference declaration and initialization (TestReferenceDeclaration.cpp) */
#include <iostream>
using namespace std;

int main() {
      int number = 88;            // Declare an int variable called number
     int & refNumber = number;
     // Declare a reference (alias) to the variable number
     // Both refNumber and number refer to the same value

     cout << number << endl;     // Print value of variable number (88)
     cout <<refNumber<< endl;    // Print value of reference (88)

     refNumber = 99;              // Re-assign a new value to refNumber
     cout <<refNumber<< endl;
     cout << number << endl;     // Value of number also changes (99)

     number = 55;                 // Re-assign a new value to number
     cout << number << endl;
     cout <<refNumber<< endl;    // Value of refNumber also changes (55)
}
```



How References Work?

A reference works as a pointer. A reference is declared as an alias of a variable. It stores the address of the variable, as illustrated:



References vs. Pointers :

Pointers and references are equivalent, except:

1. A reference is a *name constant for an address*. You need to initialize the reference during declaration.

   ```
   int & iRef;  //  Error : 'iRef' declared as reference but not initialized
   ```

   Once a reference is established to a variable, you cannot change the reference to reference another variable.

2. To get the value pointed to by a pointer, you need to use the dereferencing operator * (e.g., if pNumber is a int pointer, *pNumber returns the value pointed to by pNumber. It is called *dereferencing* or *indirection*). To assign an address of a variable into a pointer, you need to use the address-of operator & (e.g., pNumber = &number).

   On the other hand, referencing and dereferencing are done on the references implicitly. For example, if refNumber is a reference (alias) to another int variable, refNumber returns the value of the variable. No explicit dereferencing operator * should be used. Furthermore, to assign an address of a variable to a reference variable, no address-of operator & is needed.

For Example,

```
/* References vs. Pointers (TestReferenceVsPointer.cpp) */
#include <iostream>
using namespace std;

int main() {
  int number1 = 88, number2 = 22;

   // Create a pointer pointing to number1
  int * pNumber1 = &number1;   // Explicit referencing
 *pNumber1 = 99;               // Explicit dereferencing
  cout << *pNumber1 << endl;   // 99
  cout <<&number1 << endl;     // 0x22ff18
  cout << pNumber1 << endl;
  // 0x22ff18 (content of the pointer variable - same as above)
  cout <<&pNumber1 << endl;   // 0x22ff10 (address of the pointer variable)
  pNumber1 = &number2;      // Pointer can be reassigned to store another address

   // Create a reference (alias) to number1
  int & refNumber1 = number1;   // Implicit referencing (NOT &number1)
  refNumber1 = 11;               // Implicit dereferencing (NOT *refNumber1)
  cout << refNumber1 << endl;   // 11
```

```
 cout <<&refNumber1 << endl; // 0x22ff18
//refNumber1 = &number2;      // Error! Reference cannot be re-assigned
// error: invalid conversion from 'int*' to 'int'
 refNumber1 = number2;         // refNumber1 is still an alias to number1.
// Assign value of number2 (22) to refNumber1 (and number1).
 number2++;
 cout << refNumber1 << endl;   // 22
 cout << number1 << endl;      // 22
 cout << number2 << endl;      // 23
}
```

A reference variable provides a new name to an existing variable. It is *dereferenced implicitly* and does not need the dereferencing operator * to retrieve the value referenced. On the other hand, a pointer variable stores an address. You can change the address value stored in a pointer. To retrieve the value pointed to by a pointer, you need to use the indirection operator *, which is known as *explicit dereferencing*. Reference can be treated as a const pointer. It has to be initialized during declaration, and its content cannot be changed.

Reference is closely related to pointer. In many cases, it can be used as an alternative to pointer. A reference allows you to manipulate an object using pointer, but without the pointer syntax of referencing and dereferencing.

The above example illustrates how reference works, but does not show its typical usage, which is used as the function formal parameter for pass-by-reference.

Pass-By-Reference into Functions with Reference Arguments vs. Pointer Arguments

Pass-by-Value

In C/C++, by default, arguments are passed into functions *by value* (except arrays which is treated as pointers). That is, a clone copy of the argument is made and passed into the function. Changes to the clone copy inside the function has no effect to the original argument in the caller. In other words, the called function has no access to the variables in the caller. For example,

```
/* Pass-by-value into function (TestPassByValue.cpp) */
#include <iostream>
using namespace std;
int square(int);
int main() {
      int number = 8;
      cout <<  "In main(): " <<&number << endl;  // 0x22ff1c
      cout << number << endl;          // 8
```

```
    cout << square(number) << endl; // 64
    cout << number << endl;          // 8 - no change
}

int square(int n) {   // non-const
    cout <<  "In square(): " <<&n << endl;  // 0x22ff00
    n *= n;            // clone modified inside the function
    return n;
}
```

The output clearly shows that there are two different addresses.

Pass-by-Reference with Pointer Arguments

In many situations, we may wish to modify the original copy directly (especially in passing huge object or array) to avoid the overhead of cloning. This can be done by passing a pointer of the object into the function, known as *pass-by-reference*. For example,

```
/* Pass-by-reference using pointer (TestPassByPointer.cpp) */
#include <iostream>
using namespace std;
void square(int *);
int main() {
   int number = 8;
   cout <<  "In main(): " <<&number << endl;  // 0x22ff1c
   cout << number << endl;    // 8
   square(&number);            // Explicit referencing to pass an address
   cout << number << endl;    // 64
}
void square(int * pNumber) {   // Function takes an int pointer (non-const)
        cout <<  "In square(): " << pNumber << endl;  // 0x22ff1c
*pNumber *= *pNumber;     // Explicit de-referencing to get the value pointed-to
}
```

The called function operates on the same address, and can thus modify the variable in the caller.

Pass-by-Reference with Reference Arguments

Instead of passing pointers into function, you could also pass references into function, to avoid the clumsy syntax of referencing and dereferencing. For example,

```
/* Pass-by-reference using reference (TestPassByReference.cpp) */
#include <iostream>
using namespace std;

      void square(int &);

      int main() {
         int number = 8;
         cout <<  "In main(): " <<&number << endl;  // 0x22ff1c
         cout << number << endl;  // 8
         square(number);           // Implicit referencing (without '&')
         cout << number << endl;  // 64
}

      void square(int & rNumber) { // Function takes an int reference (non-const)
         cout <<  "In square(): " <<&rNumber << endl;  // 0x22ff1c
      rNumber *= rNumber;         // Implicit de-referencing (without '*')
}
```

Again, the output shows that the called function operates on the same address, and can thus modify the caller's variable.

Take note referencing (in the caller) and dereferencing (in the function) are done implicitly. The only coding difference with pass-by-value is in the function's parameter declaration.

Recall that references are to be initialized during declaration. In the case of function formal parameter, the references are initialized when the function is invoked, to the caller's arguments.

References are primarily used in passing reference in/out of functions to allow the called function accesses variables in the caller directly.

"const" Function Reference/Pointer Parameters

A const function formal parameter cannot be modified inside the function. Use const whenever possible as it protects you from inadvertently modifying the parameter and protects you against many programming errors.

A const function parameter can receive both const and non-const argument. On the other hand, a non-const function reference/pointer parameter can only receive non-const argument. For example,

```
/* Test Function const and non-const parameter (FuncationConstParameter.cpp) */
#include <iostream>
using namespace std;

int squareConst(const int);
    int squareNonConst(int);
    int squareConstRef(const int &);
    int squareNonConstRef(int &);

    int main() {
    int number = 8;
    const int constNumber = 9;
    cout << squareConst(number) << endl;
    cout << squareConst(constNumber) << endl;
    cout << squareNonConst(number) << endl;
    cout << squareNonConst(constNumber) << endl;

    cout << squareConstRef(number) << endl;
    cout << squareConstRef(constNumber) << endl;
    cout << squareNonConstRef(number) << endl;
    // cout << squareNonConstRef(constNumber) << endl;
     // error: invalid initialization of reference of
    //   type 'int&' from expression of type 'const int'
}
    int squareConst(const int number) {
    //number *= number;// error: assignment of read-only parameter
         return number * number;
    }
int squareNonConst(int number) {   // non-const parameter
    number *= number;
    return number;
}
int squareConstRef(const int & number) {   // const reference
    return number * number;
}
int squareNonConstRef(int & number) {   // non-const reference
    return number * number;
}
```

Passing the Function's Return Value :

Passing the Return-value as Reference

You can also pass the return-value as reference or pointer. For example,

```
/* Passing back return value using reference (TestPassByReferenceReturn.cpp) */
#include <iostream>
using namespace std;

int & squareRef(int &);
int * squarePtr(int *);

int main( ) {
   int number1 = 8
   cout << "In main ( )  & number1: " <<&number1 << endl;     // 0x22ff14
   int & result = squareRef (number1) ;
   cout <<  "In main ( )  &result :  "  <<&result << endl;         // 0x22ff14
   cout << result << endl;    //  64
   cout << number1<< endl;    //  64

   int number2 = 9;
   cout <<   "In main ( )  &number2:  "  <<&number2  << endl;      //  0x22ff10
   int * pResult = squarePtr(&number2);
   cout << *pResult << endl;  // 81
   cout << number2 << endl;   // 81
}

int & squareRef(int & rNumber)   {
   cout <<  "In squareRef( ):  "  <<&rNumber  << endl;      //  0x22ff14
   cout <<  "In squareRef( ) : "  <<&rNumber  << endl;      //  0x22ff14
   return  rNumber;
}

int * squarePtr(int * pNumber)   {
   cout <<  "In squarePtr( ) :  " <<  pNumber  << endl ;      //  0x22ff10
   *pNumber  *=  *pNumber ;
   return  pNumber;
```

You should not pass Function's local variable as return value by reference

```
/* Test passing the result (TestPassResultLocal.cpp)  */
#include   <iostream>
using namespace std;

int  *  squarePtr(int) ;
int & squareRef(int) ;

int main( )   {
   int number = 8;
   cout << number << endl;     //8
   cout << *squarePtr(number)  << endl;      // ??
   cout << squareRef(number)  << endl;      // ??
}

int * squaePtr(int number)    {
   int localResult = number * number;
   return &localResult ;
//   warning: address of local variable 'localResult' returned
}

int & squareRef(int number)     {
   int localResult = number * number;
   return  localResult;
//   warning: reference of local variable 'localResult' returned
}
```

This program has a serious logical error, as local variable of function is passed back as return value by reference. Local variable has local scope within the function, and its value is destroyed after the function exits. The GCC compiler is kind enough to issue a warning (but not error).

It is safe to return a reference that is passed into the function as an argument. See earlier examples.

Passing Dynamically Allocated Memory as Return Value by Reference

Instead, you need to dynamically allocate a variable for the return value, and return it by reference.

```
/*  Test passing the result (TestPassResultNew.cpp)  */
#include  <iostream>
using  namespace  std;

int  *  squarePtr(int);
int  &  squareRef(int);
int main( )    {
   int number = 8;
   cout << number << endl;     // 8
   cout << *squarePtr(number)   <<   endl;        // 64
   cout << squareRef(number)   <<   endl;        // 64
}

int  * squarePtr(int  number)    {
   int  *  dynamicAllocatedResult  =  new  int(number  *  number);
   return  dynamicAllocatedResult;
}
int  & squareRef(int  number)    {
   int    *    dynamicAllocatedResult  =  new  int(number  *  number);
   return  *dynamicAllocatedResult;
}
```

Summary :

Pointers and references are highly complex and difficult to master. But they can greatly improve the efficiency of the programs.

For novices, avoid using pointers in your program. Improper usage can lead to serious logical bugs. However, you need to understand the syntaxes of pass-by-reference with pointers and references, because they are used in many library functions.

- In *pass-by-value*, a clone is made and passed into the function. The caller's copy cannot be modified.
- In *pass-by-reference*, a pointer is passed into the function. The caller's copy could be modified inside the function.
- In *pass-by-reference with reference arguments*, you use the variable name as the argument.
- In *pass-by-reference with pointer arguments*, you need to use &*varName* (an address) as the argument.

7 Dynamic Memory Allocation

New and Delete Operators :

Instead of define an int variable (int number), and assign the address of the variable to the int pointer (int *pNumber = &number), the storage can be dynamically allocated at runtime, via a new operator. In C++, whenever you allocate a piece of memory dynamically via new, you need to use delete to remove the storage (i.e., to return the storage to the heap).
The new operation returns a pointer to the memory allocated. The delete operator takes a pointer (pointing to the memory allocated via new) as its sole argument.

For example,

```
// Static allocation
int number = 88;
int * p1 = &number;    // Assign a “valid” address into pointer

// Dynamic Allocation
int * p2;              //  Not initialize, points to somewhere which is invalid
cout << p2 << endl;    // Print address before allocation
p2 = new int;
// Dynamically allocate an int and assign its address to pointer
// The pointer gets a valid address with memory allocated
*p2 = 99;
// The pointer gets a valid address with memory allocated
*p2 = 99;
cout << p2 << endl;    // Print address after allocation
cout << *p2 << endl;   // Print value point-to
delete p2;              // Remove the dynamically allocated storage
```

Observe that new and delete operators work on *pointer*.

To initialize the allocated memory, you can use an initializer for fundamental types, or invoke a constructor for an object. For example,

```
// use an initializer to initialize a fundamental type (such as int, double)
int * p1 = new int(88);
double * p2 = new double(1.23);

// c++11 brace initialization syntax
int * p1 = new int {88};
double * p2 = new double {1.23};

// invoke a constructor to initialize an object (such as Date, Time)
Date * date1 = new Date(1999, 1, 1);
Time * time1 = new Time(12, 34, 56);
```

You can dynamically allocate storage for *global* pointers inside a function. Dynamically allocated storage inside the function remains even after the function exits. For example,

```
// Dynamically allocate global pointers (TestDynamicAllocation.cpp)
#include <iostream>

int * p1, * p2;     // Global int pointers

//  This function allocates storage for the int*
//  which is available outside the function
void allocate( )    {
   p1 = new int;        //  Allocate memory, initially content unknown
   *p1 = 88;            //  Assign value into location pointed to by pointer
   p2 = new int(99);    //   Allocate and initialize
}

int main( )    {
   allocate( );
   cout << *p1 << endl;     //  88
   cout << *p2 << endl;     //  99
   delete p1;     // Deallocate
   delete p2;
   return 0;
}
```

The main differences between static allocation and dynamic allocations are:

1. In static allocation, the compiler allocates and deallocates the storage automatically, and handle memory management. Whereas in dynamic allocation, you, as the programmer, handle the memory allocation and deallocation yourself (via new and delete operators). You have full control on the pointer addresses and their contents, as well as memory management.
2. Static allocated entities are manipulated through named variables. Dynamic allocated entities are handled through pointers.

new[] and delete[] Operators :

Dynamic array is allocated at runtime rather than compile-time, via the new[] operator. To remove the storage, you need to use the delete[] operator (instead of simply delete). For example,

```
/*  Test dynamic allocation of array    (TestDynamicArray.cpp)   */
#include <iostream>
#include <cstdlib>
using namespace std;
int main( )   {
   const int SIZE = 5;
   int * pArray;
   pArray = new int[SIZE];       // Allocate array via new[ ] operator
// Assign random numbers between 0 and 99
   for (int i = 0; i < SIZE; ++i)   {
      *(pArray + 1) = rand( ) % 100;
  }
//  Print array
  for (int i = 0; i < SIZE;  ++i)   {
     cout << *(pArray + 1) << "  ";
  }
  cout << endl;
  delete[ ] pArray;       // Deallocate array via delete[ ] operator
  return 0;
}
```

C++03 does not allow your to initialize the dynamically-allocated array. C++11 does with the brace initialization, as follows:

```
// c++11
int * p = new int[5]  {1, 2, 3, 4, 5};
```

8 Time & Space Complexiety

In the world of computing, we want our programs to be fast and efficient. But fast and efficient are loose terms.

A software might be fast on a supercomputer but its insanely slow on a personal computer. So to define this *fast*ness and *efficient*ness, we take the help of time and space complexity analysis.

Complexity analysis is a way to quantify or *measure* time and space required by an algorithm with respect to the input fetched to it.

In time complexity analysis, we are concerned with this growth which remains same on all machines. This analysis doesn't require algorithms to be run. It can be found just by doing little maths.

```
int print(int n){
    for(int i = 0; i < 10; ++i){
        cout << "Coding Blocks";
    }
}
```

Irrespective of n, print function runs exactly 10 times. So, we ll say, time complexity is T(N) = O(1). This is Big-O notation

T(N) here denotes the time complexity when represented in terms of N, the input size.

```
int print(int n){
    for(int i = 0; i < n; ++i){
        cout << "Coding Blocks";
    }
}
```

As N increases linearly, printing also happens with the same order. So time complexity is proportional to N.

```
T(N) = O(N)
int print(int n){
    for(int i = 0; i < n; ++i){
        for(int j = i; j < n; ++j)
        cout << "Coding Blocks";
    }
}
```

Lets us assume n = 4

When *i* is	0	1	2	3
Then inner loop runs	4	3	2	1

Total printing done = Total time inner loop runs = Sum of first 4 natural numbers

T(N) = Sum of first N natural Numbers = N (N + 1) / 2 = N^2 / 2 + N / 2

To get the Big-O notation, just keep the term with the highest power of N, $N^2/2$ in our case

Remove all constants associated with this term, in our case 1 / 2 will be removed

Remaining term is the proportionality factor. Hence, we will say time complexity is Big-O of N square.

T(N) = O(N^2)

Constant Complexity: O(1)

A constant task's run time won't change no matter what the input value is. Consider a function that prints a value in an array.

```
void print_element(int arr[], int idx){
      cout << "arr[" << idx << "] " << idx << " " << arr[i];
}
```

No matter which element's value you're asking the function to print, only one step is required. So we can say the function runs in O(1) time; its run-time does not increase. Its order of magnitude is always 1.

Linear Complexity: O(n)

A linear task's run time will vary depending on it's input value. If you ask a function to print all the items in a 10-element array, it will require less steps to complete than it would a 10,000 element array. This is said to run at O(n); it's run time increases at an order of magnitude proportional to n.

```
void print_array(int arr[]){
    int i = 0;
    while(arr[i]){
        cout << arr[i] << " ";
        ++i;
    }
}
```

Quadratic Complexity: O(N^2)

A quadratic task requires a number of steps equal to the square of it's input value. Lets look at a function that takes an array and N as it's input values where N is the number of values in the array. If I use a nested loop both of which use N as it's limit condition, and I ask the function to print the array's contents, the function will perform N rounds, each round printing N lines for a total of N^2 print steps.

Let's look at that practically. Assume the index length N of an array is 10. If the function prints the contents of it's array in a nested-loop, it will perform 10 rounds, each round printing 10 lines for a total of 100 print steps. This is said to run in O(N^2) time; it's total run time increases at an order of magnitude proportional to N^2.

```
void print_array(int arr[], int n){
    int x = 1;
    for(int i = 0; i < size; ++i){
        for(int j = 0; j < size; ++j){
            cout << x << " " << arr[j] << " ";
        }
        ++x;
        cout << endl;
    }
}
```

Exponential: O(2^N)

O(2^N) is just one example of exponential growth (among O(3^N), O(4^N), etc.). Time complexity at an exponential rate means that with each step the function performs, it's subsequent step will take longer by an order of magnitude equivalent to a factor of N. For instance, with a function whose step-time doubles with each subsequent step, it is said to have a complexity of O(2^N). A function whose step-time triples with each iteration is said to have a complexity of O(3^N) and so on.

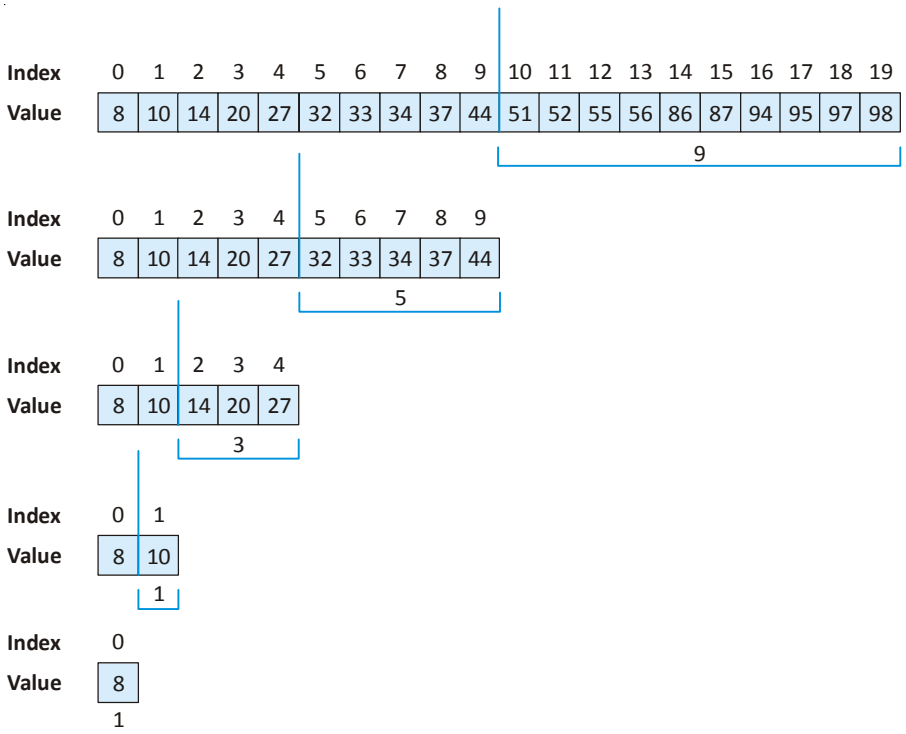
Logarithmic Complexity: O(log n)

This is the type of algorithm that makes computation blazingly fast. Instead of increasing the time it takes to perform each subsequent step, the time is decreased at magnitude inversely proportional to N.

Let's say we want to search a database for a particular number. In the data set below, we want to search 20 numbers for the number 100. In this example, searching through 20 numbers is a non-issue. But imagine we're dealing with data sets that store millions of users' profile information. Searching through each index value from beginning to end would be ridiculously inefficient. Especially if it had to be done multiple times.

A logarithmic algorithm that performs a binary search looks through only half of an increasingly smaller data set per step.

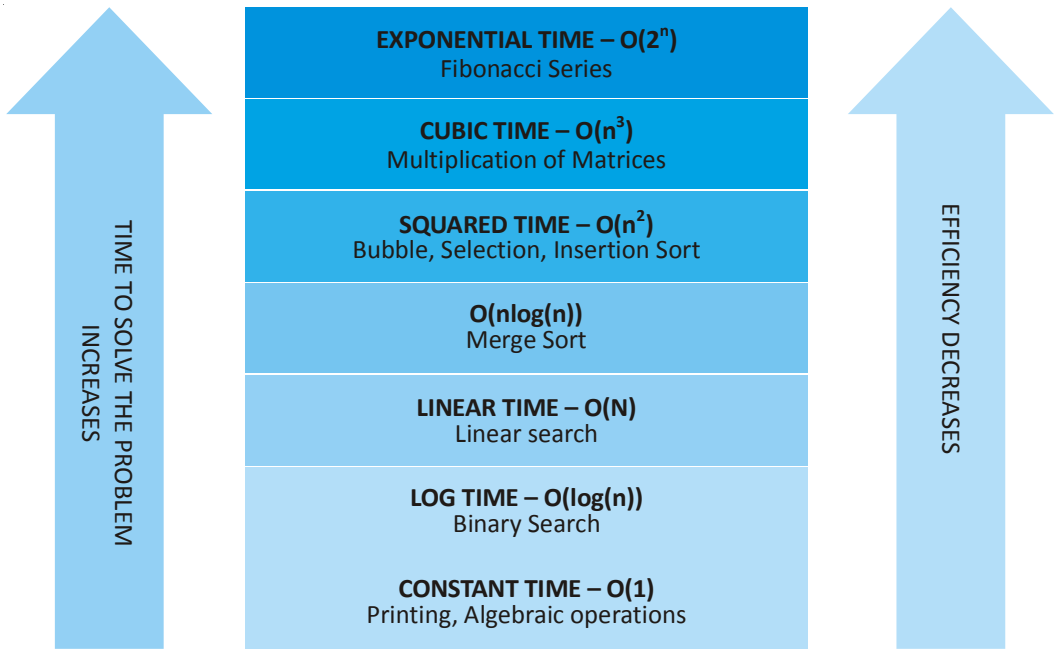
Assume we have an ascending ordered set of numbers. The algorithim starts by searching half of the entire data set. If it doesn't find the number, it discards the set just checked and then searches half of the remaining set of numbers.



As illustrated above, each round of searching consists of a smaller data set than the previous, decreasing the time each subsequent round is performed. This makes log n algorithms very scalable.

```
void binary_search(int arr[], int size, int key){
    int startIndex = 0;
    int endIndex = size - 1;
    while(startIndex <= endIndex){
        int mid = (startIndex + endIndex) / 2;
        if (arr[mid] == key) {
            return key;
        } else if(arr[mid] > key){
            // search in the left
            endIndex = mid - 1;
        }
        else {
            // search in the right
```

```
            startIndex = mid + 1;
        }
    }
    return -1;
}
```



Space Complexity :

Total amount of computer memory required for the execution of an algorithm is when expressed in terms of input size is called space complexity. Auxiliary space is the extra space or temporary space used by an algorithm. Space complexity only includes auxiliary space and NOT the space required by the input.

9 Object Oriented Programming

Programming languages help a programmer to translate his ideas into a form that the machine can understand. The method of designing and implementing the programs using the features of a language is a programming paradigm. One such method is the Procedural programming which focuses more on procedure than on data. It separates the functions and the data that uses those functions which is not a very useful thing as it makes our code less maintainable. This leads us to another technique called the object oriented programming.

What is OOPs ?

Object means any real world entity like pen, car, chair, etc. Object oriented programming is the programming model that focuses more on the objects than on the procedure for doing it. Thus we can say that object oriented programing focuses more on data than on logic or action. In this method the program is made in a way as real world works. In short it is a method to design a program using objects and classes.

Advantages of OOPs :

1. It is more appropriate for real world problems.
2. OOP provides better modularity
   Data hiding
   Abstraction
3. OOPs provides better reusability
   Inheritance
4. OOP makes our code easy to maintain and modify.
   Polymorphism

Concepts in OOPs : One of the major shortcomings of the procedural method is that the data and the functions that use those data are separated. This is not preferred as if a data member I changed then all the functions relating to it have to be changed again and again. This leads to mistakes and maintaining the code becomes tough.

How to solve?????

The best way to solve this problem would be to wrap the data and all the functions related to them in one unit. Each of this unit is an object.

Objects are the basic units of Object-oriented programming. Objects are the real world entities about which we code. Objects have properties and behavior. For instance take an example of a car which has properties like its model and has four tyres and behavior of changing speed and applying brakes.

State of the objects is represented by data members and their behavior is represented by methods.

Now many objects share common properties and behavior. So a group of such properties and behavior is made that can generate many instances of itself that have similar characteristics. This group is called a class.

Classes are the blueprints from which objects are created. Like there are many cars which share common properties and behavior. The class provides these properties and behavior to its objects which are the instances of that class.

For example a vehicle class might look like this:

```
class Vehicle {
    double price;
    int numWheels;
    int yearofManufacture;
public:
    char brand[100];
    char model[100];
    double getPrice(){
        return price;
    }
    void printDescription(){
        cout << brand << " " << model << " " << price << " " << " " <<
numWheels << endl;
    }
};
```

Now each vehicle will be a specific copy of this template.

A class in C++ contains:

- Data members
- Methods
- Constructor

Now let us talk about each of these in detail:

Data Members : Data members are the properties that are present in a class. The type of these properties can be modified by the use of special keywords called modifiers. Let us build our own student class and learn about them.

Static and Non Static Properties : Static properties are those that are common to all objects and belong to the class rather each specific object. So each object that we create doesn't have their copy. They are shared by all the objects of the class. We need to write the static keyword before it in order to make it static.

For e.g.: **static int** numStudents;

Here the number of students in a batch is a property that isn't specific to each student and hence is static.

But the properties like name, Roll Number etc can have different values for each student and are object specific and thus are non static.

An important point to note is that whenever we create a new object only the non static data member copies are created and the static properties are stored within the class only! This could be considered a very memory efficient practice as static members of a class are made only once.

A general student class might look like this:

```
class Student{
    static int numStudents;
    char name[10];
    int rollNo;
};
```

Access Modifiers :

Private: If we make any data member as private it is visible only within the class i.e. it can be accessed by and through the methods of the same class. So we can provide setters and getters function through which they can be accessed outside the class.

For instance in our student class we would like to keep the roll numbers for each student as private as we may not want any other person to modify those roll numbers by making them public. So we will make these private and provide the getter method to access the roll numbers of the students outside the student class.

```
class Student {
  private: int rollNo;
  public: int qetRollNo( ) {
     return this.rollNo;
  }
}
```

Public: It is accessible everywhere.

An important point to note here is that it is better to make a variable private and then provide getters and setters in case we wish allow others to view and change it than making the variable public. Because by providing setter we can actually add constraints to the function and update value only if they are satisfied (say if we make the marks of a student public someone can even set them to incorrect values like negative numbers. So it would be wise to provide a setter method for marks of the student so that these conditions are checked and correct marks are updated).

Methods : After having discussed about the data members of a class let us move onto the methods contained in the class relating to the data members of the class. We made many members of the class as private or protected. Now to set, modify or get the values of those data members, public getter and setter methods can be made and called on the objects of that class. The methods are called on the object name by using the dot operator.

Now let us look at various modifiers that can be used to modify the types of methods and the differences between them.

Static v/s Non Static Methods : Like data members, methods of a class can also be static which means those methods belong to the class rather than the objects for the class. These methods are directly called by the class name.

As the static methods belong to a class we don't need any instance of a class to access them. An important implication of this point is that the non static properties thus can't be accessed by the static methods as there is no specific instance of the class associated with them (the non static properties are specific to each object). So, non static members and the 'this' keyword can't be used with the static functions. Thus these methods are generally used for the static properties of the class only!

The non static methods on the other hand are called on an instance of a class or an object and can thus access both static and non static properties present in the object.

The access modifiers work the same with the methods as they do with the data members. The public methods can be accessed anywhere whereas the private methods are available only within the same class. Thus private methods can be used to work with the data members that we don't wish to expose to the clients.

Now let us add some methods to our student class.

```
class  Student{
    static int numStudents;
    char name[10];
    int rollNo;
public:
    char * getName(){
        return name;
```

```
    }
    int getRollNo(){
        return this.rollNo;
    }
    int getNumStudents(){
        return numStudents;
    }
}
```

Constructor : As we have our student class ready, we can now create its objects. Each student will be a specific copy of this template. The syntax to create an object is:

```
int main(){
    Student s;    //s is created in the memory

}
```

The method called constructor, is a special method used to initialize a new object and its name is same as that of the class name.

Even though in our student class we haven't created an explicit constructor there is a default constructor implicitly there. Every class has a constructor. If we do not explicitly write a constructor for a class, the C++ compiler builds a default constructor for that class.

We can also create our own constructors. One important point to note here is that as soon as we create our constructor the default constructor goes off. We can also make multiple constructors each varying in the number of arguments being passed (i.e. constructor overloading). The constructor that will be called will be decided on runtime depending on the type and number of arguments specified while creating the object.

Below is our own custom constructor for our student class.

```
    class Student{
        char name[10];
        int rollNo;

        Student(char n[]){
            strcpy(name, n);
        }
        Student(char name[], int rollNo){
            strcpy(this->name, n);
            this->rollNo = r;
        }
    }
```

The 'this' keyword : Here **this** is a keyword that refers to current object. So, **this.name** refers to the data member (i.e. name) of this object and not the argument variable name.

In general, there can be many uses of the 'this' keyword.

1. The 'this' keyword can be used to refer current class instance variable.
2. The this() can be used to invoke current class constructor.
3. The 'this' keyword can be used to invoke current class method (implicitly)
4. The 'this' can be passed as an argument in the method call.
5. The 'this' can be passed as argument in the constructor call.
6. The 'this' keyword can also be used to return the current class instance.

There is also a special type of constructor called the **Copy Constructor**. C++ doesn't have a default copy constructor but we can create one of our own. Given below is an example of a copy constructor.

```
class  Student{
    char name[20];
    int rollNo;

    Student(char n[], int r){
        strcpy(name, n);
        rollNo = r;
    }
    Student(Student& s){
        rollNo = s.rollNo;
        strcpy(name, s.name);
        //private members of object s are accessed by class Student
    }
};
```

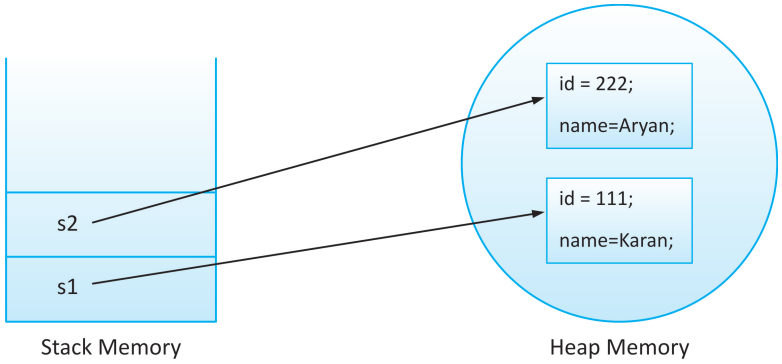
After leaning so much about the objects and constructors let us summarize the initialization of an object in simple steps.

As any new object is created, following steps take place:

1. Memory is allocated on heap and its reference is stored in stack.
2. The data members are initialized to their default values. (Only the non static properties are copied!)
3. The 'this' keyword is set to the current object.
4. The instance initialize is run and the fields are initialized to their respected values.

The constructor code is executed.

The final memory map of the instance of our student class is given below:



Encapsulation-The First Pillar of OOPs : We learnt that one of the major disadvantages of the procedural paradigm was that the functions and the data were separated which made the maintainability of the code poor! To overcome this we made classes which contained both the data members and their methods. This wrapping up of data and its functions into a single unit (called class) is known as **Encapsulation**.

A class classifies its members into three types: private, protected and public. Thus **Data Hiding** is implemented through private and protected members. These private and protected members can be accessed or set using public functions.

Also, the outside world needs to know only the essential details through public members. The rest of the implementation details remain hidden from the outside world which is nothing but **Abstraction**. Like for instance if we wish to know the aggregate marks of a student, we need not know how the aggregate is calculated. We are just interested in the marks and for that we only need to know that we need to call a public function called 'totalMarks' on any student object.

We have an additional benefit that comes bounded with encapsulation which is **Modularity**!

Modularity is nothing but partitioning our code into small individual components called modules. It makes our code less complex and easy to understand. Like for example, our code represents a school. A school has many individual components like students, teachers, other helping staff, etc. now each of them are complete units in themselves yet they are part of the school. This is called modularity.

After having the overview of benefits of encapsulation let us dive into the details of data hiding and abstraction.

Let us suppose that we make all the data members of our student class as public. Now accidently if any of our client changes the roll numbers of the student objects, all are data would be ruined. Thus public data is not safe. So, to make or data safe, we need to hide our data by making it either private or protected! This technique is called information hiding. The private data can be accessed only via a proper channel that is through their access methods which are made public.

To use a class, we only need to know the public API of the class. We only need to know the signature of the public methods that is their input and output forms to access a class. Thus, only the essential details are sown to the end user and all the complex implementation details are kept hidden. This is Abstraction. After all sometimes ignorance is bliss!

10 Linked Lists

We have used arrays to store multiple values under a single name. But using arrays has its disadvantages that :

- They are static and hence their size cannot be changed
- It needs contiguous memory to store its values
- It is difficult(costly) to insert and delete elements from an array

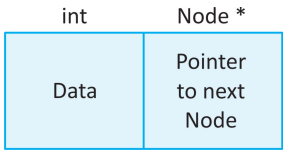
To overcome these problems, we use another data structure called linked lists.

Linked list is a linear data structure that contains sequence of elements such that each element has a reference to its next element in the sequence. Each element in a linked list is called “Node”. Each Node contains a data element and a Node next which points to the next node in the linked list. In a linked list we can create as many nodes as we want according to our requirement.

This can even be done at the runtime. The memory allocation is done at the run time and is known as DYNAMIC MEMORY ALLOCATION. Thus, linked list uses dynamic memory allocation to allocate memory at the run time.

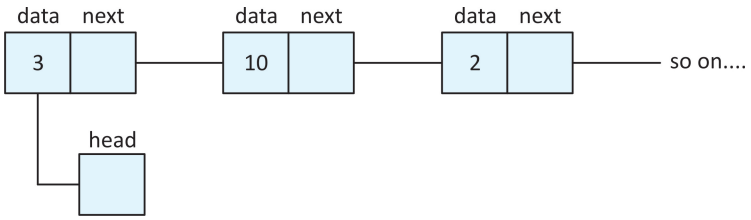
General Implementation : The linked list class contains a structure Node and a Node head. The Node head points to the starting of the linked list. Every Node contains a data element and a Node next which points to the next node in the linked list. The last node points to null.

```
struct  Node{
int data;                  //Data of the Node
Node* next; // pointer to t
}
```

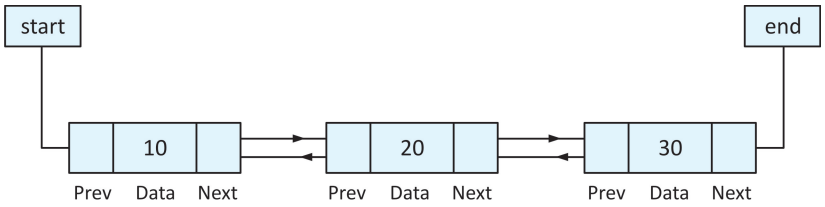


Different Types Of Linked Lists :

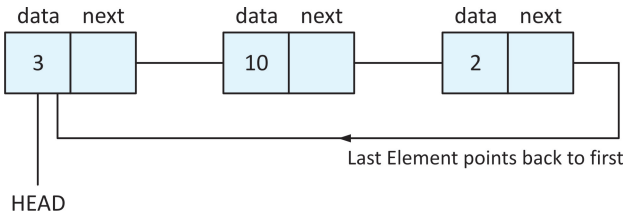
Singly Linked List : These are normal linked lists in which every node points to the next node and head node points to the starting node.



Doubly Linked List : These are linked lists in which every node points to the next node as well as the previous node.



Circular Linked List : This is a type of singly linked list in which the last node points to the starting node.



Advantages of Linked Lists :

They are dynamic in nature as their size is not fixed and can be changed during the runtime.
Insertion and deletion operations can be easily implemented at any point in the linked list.

Disadvantages of Linked Lists :

Some amount of memory is wasted in storing the references for the next nodes.
Unlike array, no element can be accessed randomly in a linked list.

Applications :

1. Process Queue's in operating system are actually doubly linked list of processes in ready state where the process at the front of linked list denotes the one to be operated next.
2. Any situation where insertion and deletion operations are more as compared to retrievals.

Big O efficiency :

Insertion	O(n)
Deletion	O(n)
Searching	O(n)

3. Reverse a linked list

```
class Node{
public:
   int data;
   Node* next;
}

Node* rev(Node* root){
   if(root is NULL || root is last node of linked list)
         return root;
   Node* temp=rev(root->next);
   root->next->next=root;
   root->next=NULL;
   return temp;
}
```

11 Stack & Queues

Stacks & Queues :

Your friend sent you 100 messages. Which message you intent to read first. If the answer is the 100th message, then you are probably using the stack without knowing it.

Our daily life make use of plenty of such examples where the last item in the series is accessed first. Other example include, pile of cash notes and plates arranges in a party.

Stack :

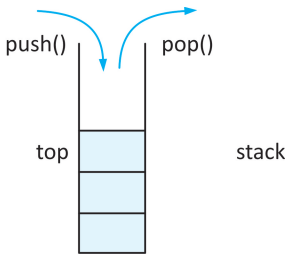
This is LIFO (Last In First Out), as the last element that is added, is the first to be removed off.

Removal from in between is not allowed, i.e. Peaking is not allowed in ideal stack.

Stack operations:

- Push : Addition of an element to stack.
- Pop : Removal of element from stack.

In addition to this there is a *top()* function in the STL stack, that is used to get the topmost element in the stack.



Three ways to implement stack

1. Use Arrays
2. Use Linked list

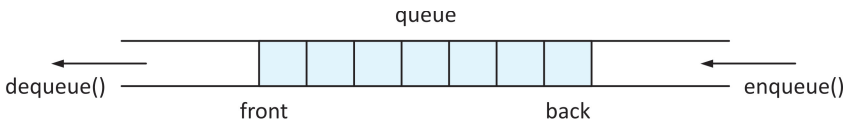
The structure of a class Stack

```
class Stack {
private:
      int array[20];
      int top;
public:
      Stack();    //constructor
      int size();
      void pop();
      void push(int x);
      bool empty();
};
```

Queue :

Suppose your friend just sent you 100 messages. If you wish to read the 1st message first, then the 2nd message and so on, then you are basically reading it in a *queue* fashion.

This follow FIFO(First In First Out) method. Another example of queue is getting a token on a railway platform. The person who came first will receive the token first. And not to foreget the queue outside ATMs after demonetization.



Functions on queue :

- **Enqueue/push** : enter an element to the queue.
- **Dequeue/pop** : remove an element from the queue.
- **IsFull** : check whether the queue is full or not.
- **IsEmpty :** check whether the queue is empty or not.
- **Size :** return the size of the array.

Queue like stack can be implemented using Linked List , arrays, or vectors. etc

12 Generic Programming in C++ : A Conceptual Overview

A computer deals with 3 things:

(a) Data **(b)** Algorithm **(c)** Container

Let us consider a simple function to search an element in an array.

Observation :

The search function above works only for an *integer array*. However the functionality *search,* is logically separate from array and applicable to all data types, i.e. searching a char in a char array or searching is applicable in searching in a linked list.

Hence, data, algorithms and containers are logically separate but very strongly connected to each other.

Generic Programming enables programmer to achieve this logical separation by writing **general** *algorithms that work on all containers with all data types.*

Separating Data

A function can be made general to all data types with the help of templates.

Templates are a blueprint based on which compiler generates a function as per the requirements.

To create a template of search function, we *replace int with a type T* and tells compiler that *T is a type* using the *statement* template <class T>

```
template <class T>
int search(T arr[], int n, T elementToSearch) {
  for (int pos = 0; pos < n; ++pos) {
     if (arr[pos] == elementToSearch) {
       return pos;
     }
  }
  return -1;
}
```

Now the search function runs for all types of arrays for which statement 4 is defined.

Now the search function runs for all types of arrays for which statement 4 is defined.

```
int arrInt[100];
char arrChar[100];
float arrFloat[100];
Book arrBook[100], X;       //X is a book
search(arrInt, 100, 5);     //T is replaced by int
search(arrChar, 100, 'A');  //T is replaced by char
search(arrFloat, 100, 1.24); //T is replaced by float
search(arrBook, 100, X); //T is replaced by Book.
```

So, one function can be run on different data types. This makes our function *general* for all types of data.

Note

In the actual code that is produced after compilation, 4 different functions will be produced based on the template with T replaced accordingly.

You could use <typename T> or <class T> in the template statement 1. The keywords typename and class serve the same purpose.

Separating Algorithm

The search function will not work for Book objects if computer doesn't know how to compare 2 books. So our function is limited in some sense. To work it for all data types, we use a concept of *comparator* (also see predicate).

Let's rewrite the templated search function again

```
template <class T, class Compare>
int search(T arr[], int n, T elementToSearch, Compare obj) {
//compare is a class that has () operator overloaded
  for (int pos = 0; pos < n; ++pos) {
      if (obj(arr[pos], elementToSearch) == true) {
//obj compares elements of type T
        return pos;
    }
  }
  return -1;
}
```

To use the search function for integers, you shall now write:

```
//defining a class compare
class compareInt {
public:
   bool operator()(int a, int b) {
      return a == b  ? true  : false;
  }
};
//calling search templated function
compareInt obj;
        //obj is the object of class compareInt
search(arrInt, 100, 20, obj); //T replaced with int
        //Compare replaced with compareInt
```

Line 4 works since obj(xInt, yInt) is defined by the class compareInt.

To use *search function for a book class*, we should write a compareBook class.

```
class compareBook{
public:
bool operator()(const Book& B1, const Book& B2){
return B1.getIsbn () == B2.getIsbn();
}

search(arrBook, 100, X, compareBook);
        //calling search function
```

The same search function now operators for Books just by writing a small compare class.

However, search function still works only for arrays. However the functionality of searching extends to list equally. To make it general for all containers(here array) we introduce a concept of iterators in our discussion.

Separating Containers

Iterators

Visualise iterators as an entity using which you can access the data within a container with certain restrictions.

These are classified into 5 categories

Input Iterator : An entity through which you can read from container and move *ahead.*

What sort of container will posses an input iterator???

A keyboard.

Output Iterator An entity through which you can write into the container and move ahead.
Container like printer or monitor will have such an iterator.

Forward Iterator Iterator with functionalities of both Input and Output iterator in single direction.
Singly linked list will posses a forward entity since we can read/write only in the forward direction.

Bidirectional Iterator Forward iterator that can move in both the directions.

Doubly linked list will posses a bidirectional iterator.

Random Access Iterator Iterator that can read/write in both directions plus can also take jumps.

An array will have random access iterator. Since, you can jump by writing arr[5], which means jump to the 5th element.

Entity that does this, behaves like a pointer in some sense.

To write search function that is truly independent of data and the underlying container, we use iterator.

```
template<class ForwardIterator, class T, class Compare>
ForwardIterator search(ForwardIter beginOfContainer, ForwardIter endOfContainer,
T elementToSearch, Compare Obj) {
   while (beginOfContainer  != endOfContainer) {
       if (obj((*beginOfContainer), elementToSearch) == true) break; //iterators are like
pointers!
     ++beginOfContainter;
  }
   return beginOfContainer;
}
```

Here, beginOfContainer is a *ForwardIterator*, i.e., beginOfContainer must know how to read/write and move in *forward* direction.

So, if a container has at least ForwardIterator, the algorithm works. Hence, it works for list, doubly linked list and array as well thus achieving generality over container.

```
//search for book in an array
search(arr, arr + n, X, compareBook);
//search for book in a list
list<Book> lb;  // see list
search(lb.begin(), lb.end(), X, compareBook);
//begin and end are member function of the class list.
```

Summary

1. *Using templates, we achieve freedom from data types*
2. *Using comparators, we achieve freedom from operation(s) acting on data*
3. *Using iterators, we achieve freedom from underlying data structure (container).*