Abstract Data Type & C++ Revision

Implementation
- choosing a particular data structure

ABSTRACT DATA TYPE(ADT)
- A collection of data and a set of operations on the data

ABSTRACTION
- Specification of each module are written before implementation
- Separate the purpose of module from its implementation

ADT is composed

A collection of data
A set of operations on that data

Abstraction and Information Hiding

Data abstraction
- Focuses on the operations of data, not on the implementation of the operations
- Asks you to think *what* you can do to a collection of data independently of *how* you do it
- Allows you to develop each data structure in relative isolation from the rest of the solution
- A natural extension of functional abstraction

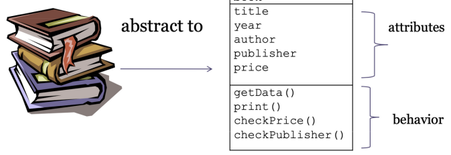
Functional abstraction
- Separates the purpose of a module from its implementation

Specification of ADT

What the ADT operations do,not how to implement them

Information hiding
- Hide details within a module
- Ensure that no other module can tamper with these hidden details
- Makes these details inaccessible from outside the module
- Public view of a module
- Described by its specifications
- Private view of a module
- Implementation details that the specifications should not describe

Abstraction of a book



Constructors
- Create and initialize new instances of a class
- Invoked when you declare an instance of the class
- Have the same name as the class
- Have no return type, not even void

A class can have several constructors
- A default constructor has no arguments
- The compiler will generate a default constructor if you do not define any constructors

Class methods/member functions

- **Constructor** – allocates memory for an object and can initialize new instances of a class to particular values.
- **Destructor** – destroys an instance of a class when the object's lifetime ends.
- C++ function
- **const function** – function that cannot alter data member of the class

C++ Classes

- An object is an instance of a class
- A class defines a new data type
- A class contains data members and methods (member functions)
- By default, all members in a class are private But can be specified as public

C++ Class definition

```
class clasName
{
public:
    list of data member declaration;
    list of function member declaration;
private:
    list of data member declaration;
    list of function member declaration;
};  // end class definition
```

class members: data and function

public : members that are accessible by other modules
private : members that are hidden from other modules and can only be accessed by function member of the same class.

Destructor
- Destroys an instance of an object when the object's lifetime ends
- Each class has one destructor
  - For many classes, you can omit the destructor
  - The compiler will generate a destructor if you do not define one
- Example:
  ~Sphere();

Classes as Function Parameter

- Pass by value
- Pass by reference

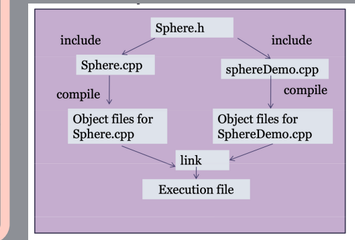
functionType functionName(className & classObject)

- Pass by const reference

functionType functionName(const className & classObject)

Encapsulation

- Encapsulation is the process of combining data and functions into a single unit called class.
- The programmer cannot directly access the data. Data is only accessible through the functions present inside the class.
- Data encapsulation led to the important concept of data hiding.
- Data hiding is the implementation details of a class that are hidden from the user. The concept of restricted access led programmers to write specialized functions or methods for performing the operations on hidden members of the class.



Compile all .cpp files separately in order to create object files. Link all files to create .exe files.

- A group of objects from the same class can be declared as array of a class
- Example:
  - Array of class students registered in Data Structure Class
  - Array of class lecturer teaching at FSKSM
  - Array of class subjects offered in Semester I
- Every element in the array of class has it's own data member and function member.
- Syntax to declare array of objects :

```
className arrayName[arraySize];
```

- Pointer – store address of a variable.
- Pointer can also store address of an object.
- Example

```
student student1; // create instance of student
student* studentPtr = &student1;
```

- Create a pointer variable **studentPtr** and initialize the pointer with the address of instance **student1**

RECURSIVE

- Recursion can be used to replace loops.
- Recursively defined data structures, like lists, are very well-suited to processing by recursive procedures and functions
- Recursive is a repetitive process in which an algorithm calls itself.

• RECURSIVE CASE(S): more complex – make use of recursion to solve smaller subproblems & combine into a solution to the larger problem

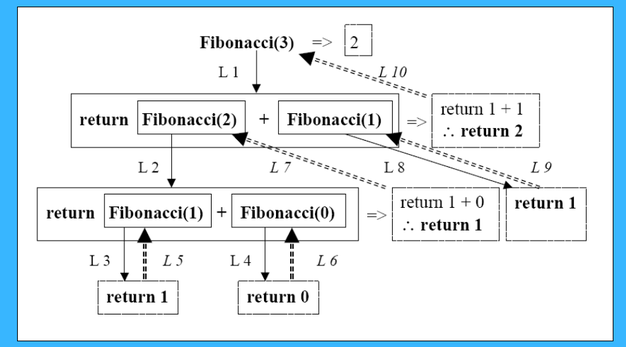
BASE CASE(S): case(s) so simple that they can be solved directly

Infinite Recursive

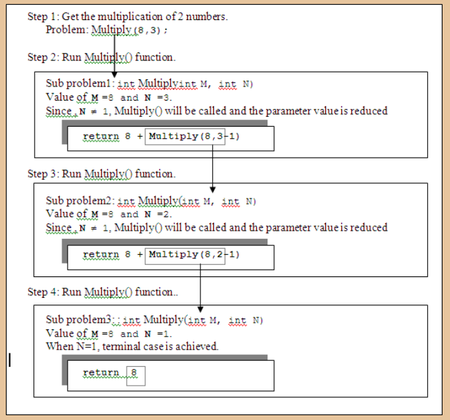
- Recursion that cannot stop is called infinite recursion
- Infinite recursion occur when the recursive function:
  - Does not has at least 1 base case (to terminate the recursive sequence)
  - Size of recursive case is not changed
  - During recursive call, size of recursive case does not get *closer to a base case*

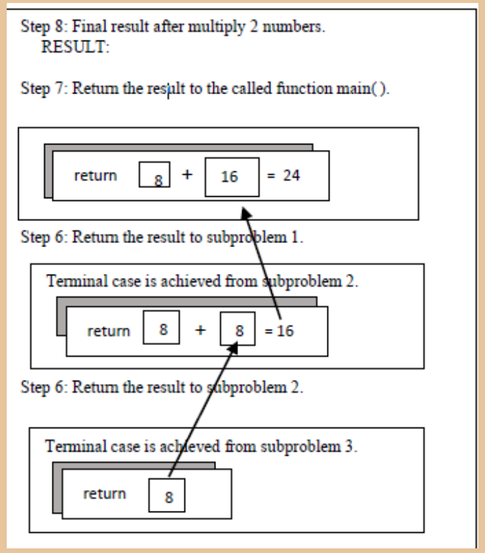
Fibonacci

```
int Fibonacci (int N )
{ /* start Fibonacci*/  if (N<=0)
return 0;  else if (N==1)
return 1;
else
return Fibonacci(N-1) + Fibonacci (N-2);
}
```



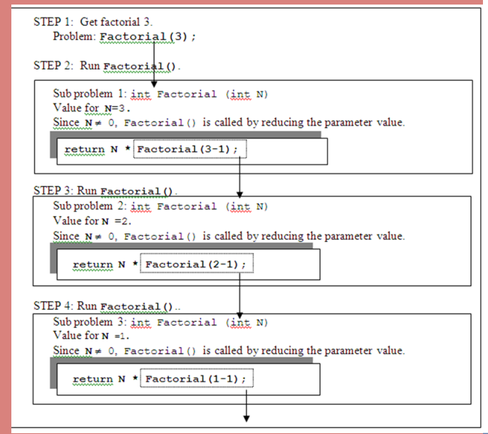
MULTIPLYING

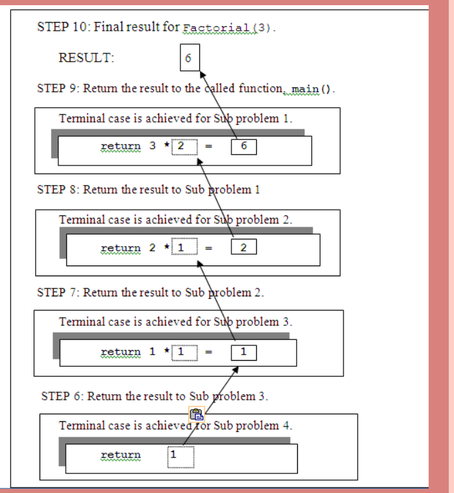




Factorial Problem

```
int Factorial (int N )
{ /*start Factorial*/  if (N==0)
return 1;
else
return N * Factorial (N-1);
} /*end Factorial
```






```
#include <stdio.h>
#include <conio.h>
void printIntegesr(int n);
main()
{
      int number;
      cout<<"\nEnter an integer value :";
      cin >> number;
      printIntegers(number);
}
void printIntegers (int nom)
{
      cout << "\Value : " << nom;
      printIntegers (nom);
}
```

1. No condition satatement to stop the recursive call.
2. Terminal case variable does not change.

Analysis of algorithms

- **Worst-case efficiency:** Longest running time for any input of size n
A determination of the maximum amount of time that an algorithm requires to solve problems of size n
- **Best-case efficiency:** Shortest running time for any input of size n
A determination of the minimum amount of time that an algorithm requires to solve problems of size n
- **Average-case efficiency:** Average running time for all inputs of size n
A determination of the average amount of time that an algorithm requires to solve problems of size n

- Complexity time can be represented by **big 'O' notation**
- Big 'O' notation is denoted as: **O(*acc*)**

  whereby:
  - ✓ **O** - order
  - ✓ ***acc*** - class of algorithm complexity
- Big O notation example:

  $O(1)$, $O(\log_x n)$, $O(n)$, $O(n \log_x n)$, $O(n^2)$

Notation	Execution time/ number of step
O(1)	Constant function, independent of input size, n. Example: Finding the first element of a list.
$O(\log_x n)$	Problem complexity increases slowly as the problem size increases. Squaring the problem size only doubles the time. Characteristic: Solve a problem by splitting into constant fractions of the problem (e.g., throw away ½ at each step)
O(n)	Problem complexity increases linearly with the size of the input, n Example: counting the elements in a list.

Notation	Execution time/ number of step
$O(n\log_x n)$	Log-linear increase - Problem complexity increases a little faster than n Characteristic: Divide problem into sub problems that are solved the same way. Example: Merge sort
$O(n^2)$	Quadratic increase. Problem complexity increases fairly fast, but still manageable Characteristic: Two nested loops of size n
$O(n^3)$	Cubic increase. Practical for small input size, n.
$O(2^n)$	Exponential increase - Increase too rapidly to be practical Problem complexity increases very fast Generally unmanageable for any meaningful n Example: Find all subsets of a set of n elements

EXAMPLE

Algorithm	Number of steps
void sample4 ( ) {	0
for (int a=2; a<=n; a++)	n – 2 + 1 = n - 1
cout << "Example of step calculation";	(n – 1).1 = n - 1
}	0
Total Steps	**2(n – 1)**

Algorithm	Number of steps
void sample6 ( ) {	0
for (int a=1; a<=n; a++)	n - 1 + 1 = n
for (int b=1; b<=n; b++)	n.(n - 1 + 1) = n.n
cout << "Example of step calculation";	n.n.1 = n.n
}	0
Total Steps	**n + $2n^2$**
Complexity Time	**$O(n^2)$**

Algorithm	Number of steps
void sample9 ( ) {	0
int n, x, i=1;	1
while (i<=n) {	$1 + \log_2 n$
x++;	$(1 + \log_2 n).1 = 1 + \log_2 n$
i=i*2;	$(1 + \log_2 n).1 = 1 + \log_2 n$
}	0
Total Steps	**$1 + 3(1 + \log_2 n)$**
Complexity Time	**$O(\log_2 n)$**

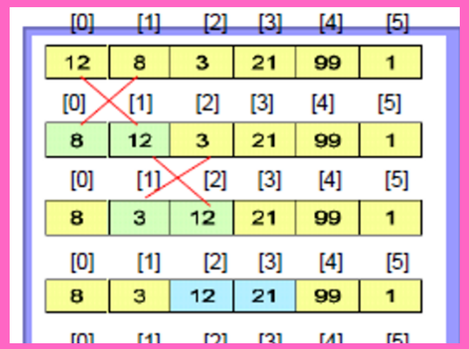
O(1) < O($\log_x$n) < O(n) < O(n $\log_2$n) < O(n^2) < O(n^3) < O(2^n)

Notasi	n = 8	n = 16	n = 32
$O(\log_2 n)$	3	4	5
$O(n)$	8	16	32
$O(n \log_2 n)$	24	64	160
$O(n^2)$	64	256	1024
$O(n^3)$	512	4096	32768
$O(2^n)$	256	65536	4294967296

Simple Sort

Bubble Sort

- Compare adjacent elements in the list
- Exchange the elements if they are out of order
- Each pass moves the largest (or smallest) elements to the end of the array
- Repeating this process eventually sorts the array into ascending (or descending) order.



Improved Bubble Sort

```
// Sorts items in an array into ascending order.
void bubbleSort (DataType data[], int n)
{    int temp;
     bool sorted = false; // false when swaps occur
     for (int pass = 1; (pass < n) && !sorted; ++pass)
     {    sorted = true; // assume sorted
          for (int x = 0; x < n-pass; ++x)
          {    if (data[x] > data[x+1])
               {    // exchange items
                    temp = data[x];
                    data[x] = data[x+1];
                    data[x+1] = temp;
                    sorted = false; // signal exchange
               } // end if
          } // end for
     } // end for
} // end bubbleSort
```

To **improve the efficiency** of Bubble Sort, a condition that check whether the list is sorted should be add at the external loop.

A Boolean variable, **sorted** is added in the algorithm to signal whether there is any exchange of elements occur in certain pass.

In external loop, **sorted** is set to **true**. If there is exchange of data inside the inner loop, sorted is set to **false**.

Another pass will continue, if sorted is false and will stop if sorted is true.

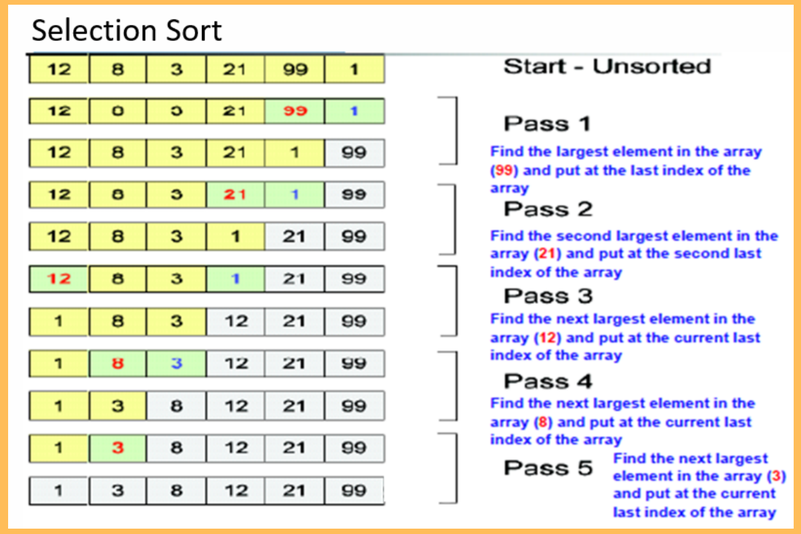
Insertion Sort

- Strategy
- Take multiple passes over the array
- Partition the array into two regions: sorted and unsorted
- Take each item from the unsorted region and insert it into its correct order in the sorted region
- Find next unsorted element and insert it in correct place, relative to the ones already sorted
- Analysis
- Appropriate for small arrays due to its simplicity

Types of Sort	Insert Sort	Bubble Sort	Selection Sort
Number of Comparisons			
Best Case	O(n)	$O(n^2)$	$O(n^2)$
Average Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n^2)$
Number of Swaps			
Best Case	0	0	O(n)
Average Case	$O(n^2)$	$O(n^2)$	O(n)
Worst Case	$O(n^2)$	$O(n^2)$	O(n)

Selection Sort

- Strategy
- Choose the largest / smallest item in the array and place the item in its correct place
- Choose the next largest / next smallest item in the array and place the item in its correct place.
- Repeat the process until all items are sorted.
- Does not depend on the initial arrangement of the data
- Only appropriate for small n - O(n2) algorithm



- Time Complexity for Selection Sort is the same for all cases - worse case, best case or average case O(n2). The number of comparisons between elements is the same.

- The efficiency of Selection Sort does not depend on the initial arrangement of the data.

Advanced Sort

- **Divide**
  - **Break into sub-problems** that are themselves **smaller instances** of the **same type** of problem
  - **Recursively solving** this problem
- **Conquer** (overcome)
  - The **solution to the original problem** is then formed from the **solutions to the sub-problems**.

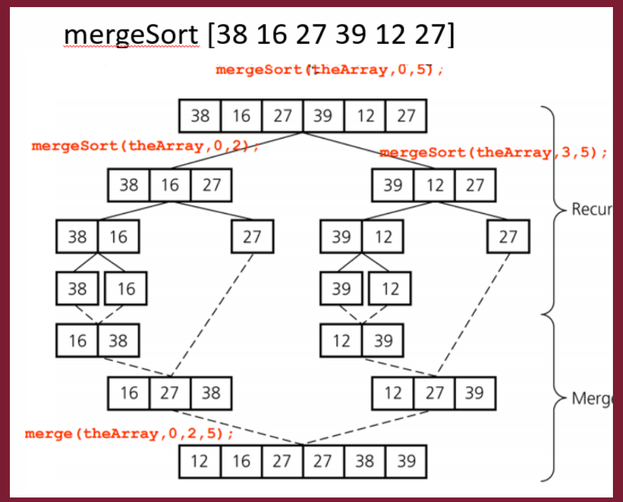
- Differences of Quick sort and Merge sort :

Quick Sort	Merge Sort
Partition the list based on the **pivot** value	Partition the list by **dividing the list into two**
No merge operation is needed since when there is only **one item left in the list to be sorted**, all **other items are already in sorted position.**	**Merge operation** is needed to **sort and merge** the item in the **left and right segments.**

Merge Sort

•Applies divide and conquer strategy.
•Three main steps in Merge Sort algorithm:
•Divide an array into halves
•Sort each half
•Merge the sorted halves into one sorted array
•A recursive sorting algorithm
•Performance is independent of the initial order of the array items

Need 2 functions
•MergeSort() function
•A Recursive function that divide the array into pieces until each piece contain only one item.
•The small pieces is merge into larger sorted pieces until one sorted array is achieved.
•Merge() function
•Compares an item into one half of the array with item in the other half of the array and moves the smaller item into temporary array. Then, the remaining items are simply moved to the temporary array. The temporary array is copied back into the original array.



Mergesort

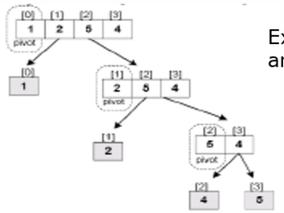
- **Analysis**
  - **Worst case: O(n * $\log_2$n)**
  - **Average case: O(n * $\log_2$n)**
    - Performance is independent of the **initial order** of the array items
- **Advantage**
  - Mergesort is an extremely **fast** algorithm
- **Disadvantage**
  - Mergesort requires a second array (**temporary** array) **as large as** the **original** array

Quick Sort

•A divide-and-conquer algorithm
•Strategy
•Choose a pivot (first element in the array)
•Partition the array about the pivot
•items < pivot
•items >= pivot
•Pivot is now in correct sorted position
•Sort the left section again until there is one item left
•Sort the right section again until there is one item left

Quick Sort Analysis

- The **efficiency** of quick sort depends on the **pivot** value.
- This class chose the **first element in the array** as pivot value.
- However, pivot can also be chosen at **random**, or from the **last** element in the array.
- The **worse case** for quick sort occur when the **smallest item or the largest item always be chosen as pivot** value causing the left partition and the right partition **not balance**.



Example of worse case quick sort: sorted array [1 2 5 4] causing imbalance partition.

- **Analysis**
  - **Average case: O(n * $\log_2$n)**
  - **Worst case: O(n^2)**
    - When the array is already sorted, and the smallest item is chosen as the pivot
- Quicksort is usually extremely fast in practice
- Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays