C/C++ Programming Language

Introduction to Programming Using C/C++

Java™ Education and Technology Services

Invest In Yourself ,
Develop Your Career

Course Duration and Evaluation

- Duration: 60 hours
  - 10 Lectures (30 hours)
  - 10 Labs (30 hours)
- Evaluation Criteria:
  - 40% on labs activities and assignments
  - 60% on written exam after 7 days of the last lectures.

Course Content

- D1: Introduction To Structured Programming Using C
- D2: Control Statements, Arrays, and String
- D3: Structures and Functions
- D4: Pointers and Dynamic Allocations
- D5: OOP Concepts, and Terminologies
- D6: Polymorphism (Function Overloading), Static Members, and Friend Function
- D7: Copy Constructor, and Polymorphism (Operator Overloading)
- D8: Association (has-A): Composition and Aggregation
- D9: Inheritance (is-A) Types, Multi-Level, and Polymorphism (Overriding)
- D10: Virtual Function, Dynamic Binding, Template Class, and Introduction to UML

Introduction to Structured Programming Using C

Java™ Education and Technology Services

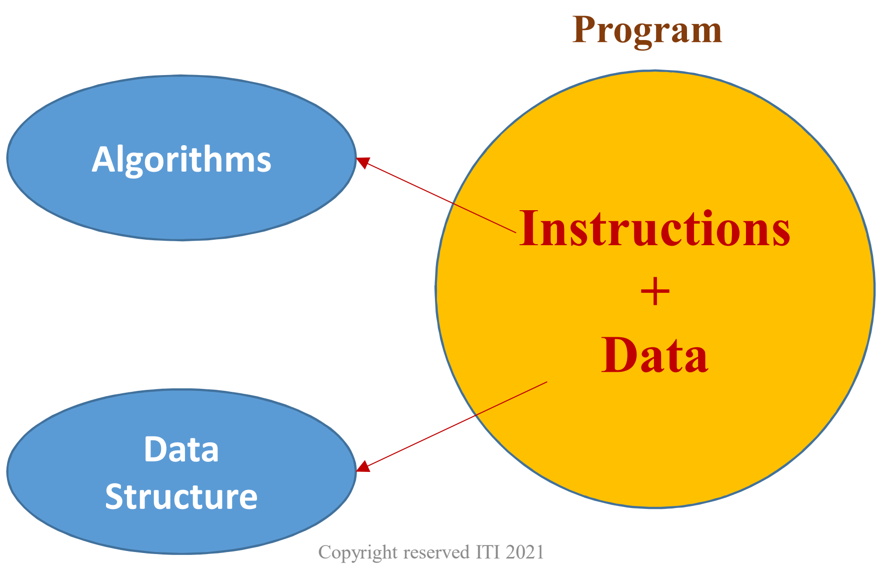
Invest In Yourself , Develop Your Career

Content

- Introduction
- Programming Languages Levels
- Programming Techniques History
- How Do Programming Languages Work?
- C Program Structure
- Some of basics in C Program
- Operators in C

1.1. Introduction

- Why we need a computer program?
  - To perform a desired task; Problem Solving
- What is the computer program includes?
  - Instructions
  - Data



1.2. Programming Languages Levels

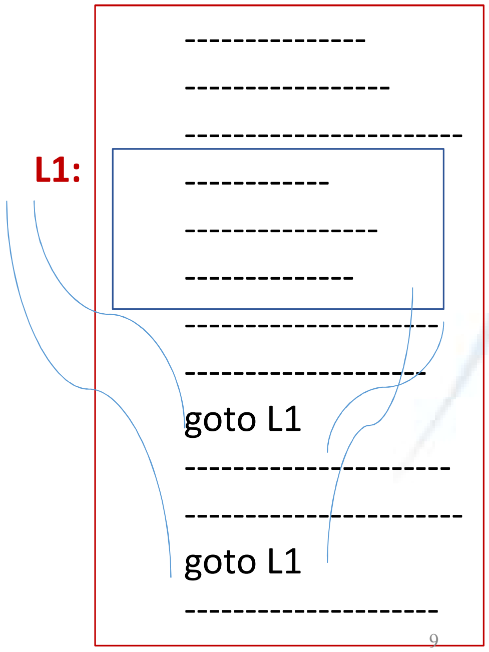
- Low-Level Languages
  - Fundamentals language for computer processors,
  - 0's and 1's that represent high and low electrical voltage (Machine Language),
  - Modified to use symbolic operation code to represent the machine operation code (Assembly Language)
- High-Level Languages
  - Use English like statements, executed by operating system (in most cases)
  - C, C++, Pascal, Java, ….
- Very High-Level Languages
  - Usually *domain-specific* languages, limited to a very specific application, purpose, or type of task, and they are often scripting languages
  - PLAN, Prolog, LISP

1.3. Programming Techniques History

- Linear Programming Languages
  - BASIC
- Structured Programming Languages
  - C, Pascal
- Object-Oriented Programming Languages
  - C++, Java, C#

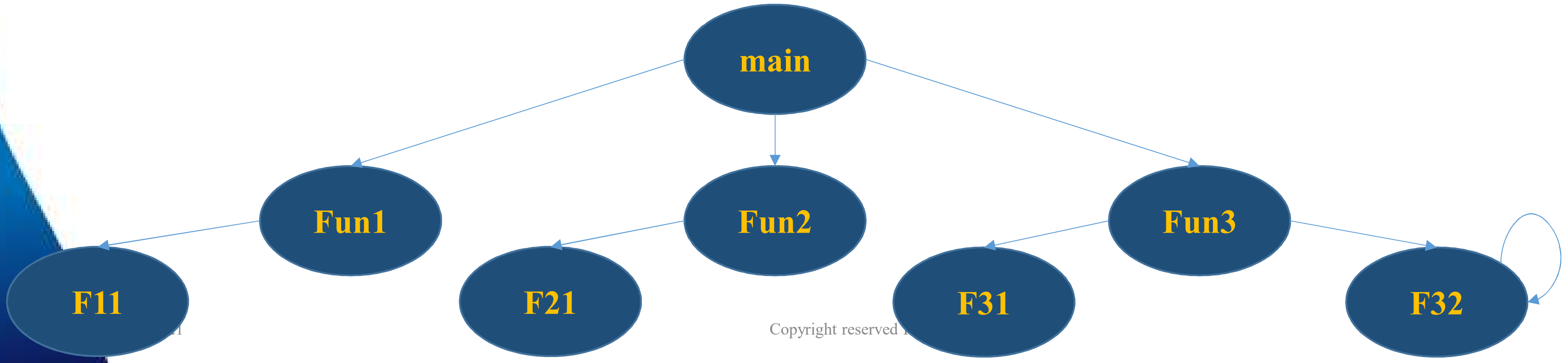
1.3.1 Linear Programming Languages

- A program is executed in a sequential manner
- The program size is exponential increasing by increasing the functionality of the program.
- GOTO statement is used …!!!



1.3.2 Structured Programming Languages

- Some times called Function-Oriented Programming
- Similar like subroutine in assembly language
- Subroutine equivalent Function equivalent Procedure ; Sub-program
- Any program consists of functions, at least “main” function
- The main function is the entry point of the program



1.3.3 Object-Oriented Programming Languages

- Programming model that organizes software design around data, or objects, rather than functions.
- An object can be defined as a data field that has unique attributes and behavior.
- A Class is the template of objects that describe the same data but each one has different values of attributes.
- Reusability of the classes is one of the advantages of this technique.
- Details will be clear in Object-Oriented Programming using C++ sections …

1.4. How Do Programming Languages Work?

- **Interpreted Languages**
  - It depends on an interpreter program that reads the source code and translates it on the fly into computations and system calls- line by line.
  - The source has to be re-interpreted (and the interpreter present) each time the code is executed. **Basic** and most of scripting languages (**HTML**)
- **Compiled Languages**
  - Compiled languages get translated into executable file, no need to recompile again –if there is no changes – run the executable directly. **C, C++, Pascal**
- **Compiled & Interpreted Languages**
  - Compiler translate to intermediate code and need the interpreter to run that intermediate code. **Java**

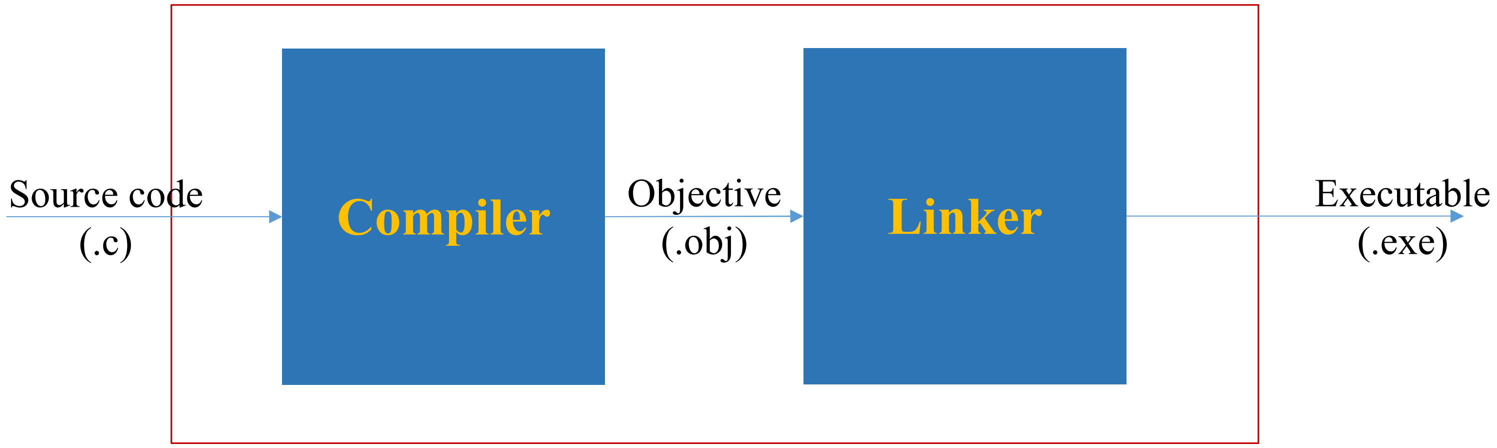
1.5. C Program Structure

- **<u>Part I</u>: Preprocessing Part**
  - Include Libraries
  - Define and Macro (Text Replacement)
  - Structures definitions
  - Global Variables
  - User-defined Functions Prototypes
- **<u>Part II</u>: The Main Entry Point : main Function**
- **<u>Part III</u>: User-defined Functions**

1.5.1.1 Include Libraries

- **#include**
- For using a C function in any program you have to include its library before using. *Why it is in the preprocessing part?*
- The compiler needs to check if you call the function in a right way or not: check on the name and the input parameters (numbers and types).
- Each library consists of C (.c)file and Header File (.h), it is prefer to include the header file not the C file. *Why?*
- The compiler check and translate the source code (.c) to executable instructions – system calls- except the predefined functions in libraries and generate intermediate file called Objective file (.obj).
- The linker program – as a part of compiler environment – use the libraries –which are included in the program - to complete the Objective file (.obj) to be an executable file. *How?*

1.5.1.1 Include Libraries



- You can include just one library using one include statement
- Must be the first of the file
- Has two forms:
  - #include <stdio.h>
  - #include “d:\\myNewLib\\extern.h”

1.5.1.2 Define and Macro

- **#define**
- It is used to define a replacement text with another after the statement directly till the end of the file before starting the compilation.
- Ex:
  - #define PI 3.14
  - ----
  - printf (“%f”, PI); // the replacement done before start compilation the o/p is “3.14”
- Ex:
  - #define ONE 1
  - #define TWO ONE+ONE
  - #define FOUR TWO*TWO

  - printf (“%d”, FOUR); // what will the output?

1.5.1.2 Define and Macro

- Macro is another type of text replacement but with using the function operator ().
- Ex:
  - #define SUM(X , Y)   X+Y
  - -------
  - -------
  - printf("%d", SUM(3, 8));   // output will be: 11
  - printf("%d", SUM(-4, 3));   // output will be: -1

1.5.1.3 Structures definitions

- Structure mean record with number of fields which they are non-homogeneous in most cases
- It will be clear in the 3rd lecture.

1.5.1.4 Global Variables

- The scope of them is global; these variables are accessible by all the functions of the program.
- It is not preferable to use it without strong reasons; use it if and only if you have to use it. *Why?*
- They are saved in a part of the memory sections allocated to the program, this part is called: ***Heap Memory***

1.5.1.5 User-defined Functions Prototypes

- It is the header only of the user-defined function in the program.
- The compiler use it to check if u call these function in a right way using tis prototype – the same concept of using the header files of the libraries-
- **Note**: you may not use this part but you have to rearrange the sequence of user-defined function and so the main function; the <u>called function </u>must be written *before* the <u>caller function.</u>

1.5.2 Part II: The Main Entry Point: main function

- At the first use the following main function header:

```
void main (void)        // The function header
{                       // Start of the function body


}                       // End of the function body
```

- First *void* indicate there is no return data type for the caller of the main function. *Whom?*
- The *void* between braces indicate there is no input parameters will be sent to the main function when it is called.

1.5.3 Part III: User-defined Functions

- It is the part to write the structure of functions you designed it to your program.
- It is the full functions; Header and body for each one.

1.6. Some of basics in C Program

- Case sensitive.
- Braces, and blocks: {}
- Delimiters after each statements – except include and define: ;
- Basic input and output functions:
  - printf
  - scanf
  - getch or getchar
  - clrscr or system("cls") ---------> compiler dependent

1.6. Some of basics in C Program

- Primitive Data Types in C:

• char	size	1	byte	from	0	to	$2^8 - 1$	unsigned
• int	size	2	bytes	from	-2^{15}	to	$2^{15}-1$	signed
• long	size	4	bytes	from	-2^{31}	to	$2^{31}-1$	signed
• float*	size	4	bytes	from	$-1.2*10^{38}$	to	$3.4*10^{38}$	signed
• double*	size	8	bytes	from	$-2.3*10^{308}$	to	$1.7*10^{308}$	signed

- Complex Data Types in C:
  - Arrays
  - Structure

* The ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic ("the IEEE standard" or "IEEE 754“)

1.7. Welcome To ITI World Program

- First program using C:

```
#include <stdio.h>
#include <conio.h>
void main (void)
{
    system(“cls”); // or use clrscr(); (compiler dependent)
    printf(“Welcome to ITI World”);
    getch();
}
```

1.8 Operators in C

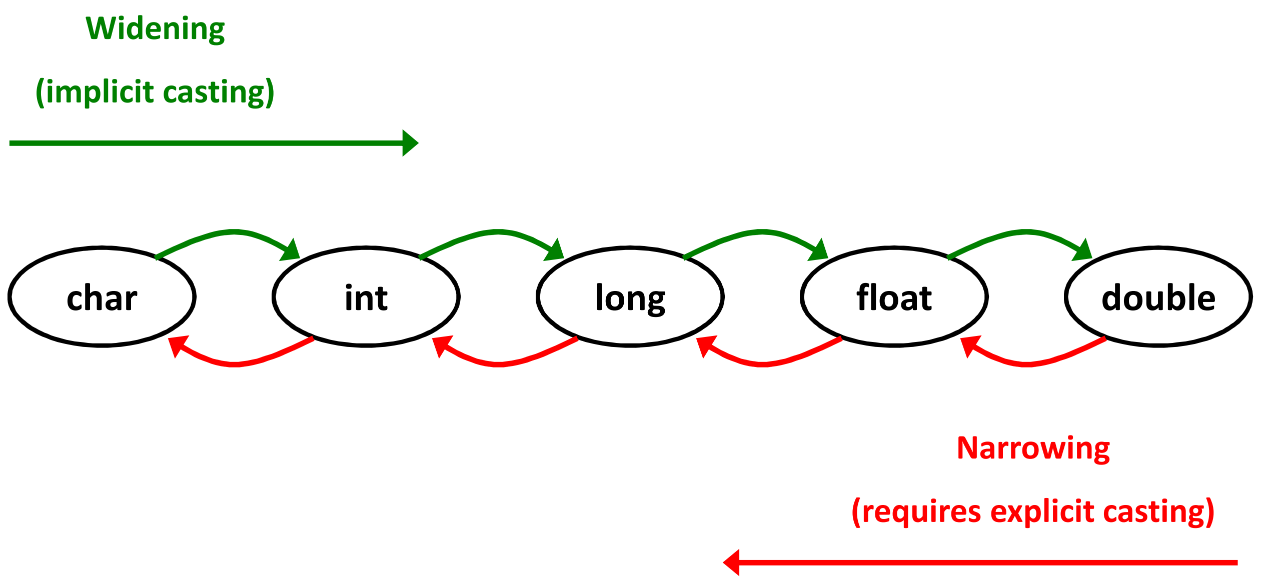
- An operator is a symbol that tells the compiler to perform a specific mathematical or logical operation.
- C language is rich in built-in operators and provides the following types of operators:
  - Unary Operators
  - Arithmetic Operators
  - Relational (Comparison) Operators
  - Bitwise (Logical) Operators
  - Short-Circuit Operators
  - Shift Operators
  - Assignment Operators
  - Misc (*miscellaneous*) Operators

1.8.1 Unary Operators

- It is operators work on just one operand, as follow:
  - + : positive sign
  - - : negative sign
  - ++ : increment operator to increase the value of the operand by 1 unit
  - -- : decrement operator to decrease the value of the operand by 1 unit
  - ! : Boolean inversion operator; !false=true, !true=false ; !0=1, !1=0 →(in C)
  - ~ : one's complement operator; convert to the one's complement of the operand; 0's → 1's and 1's → 0's
  - () : casting operator; to change the value of the operand to put it in different data type format

```
Ex:  int x=5;    int y;
y = x++;           //   y = 5     and     x = 6
y = ++x;           //   y = 6     and     x = 6
Ex:  int x=3, y=5; float f;
f = y / x;         // f = 1.0;
f = (float)y/x;    // f = 1.66666
```

Implicitly (Auto Conversion) and Explicitly casting



1.8.2 Arithmetic Operators

- The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20 then

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0

1.8.2 Arithmetic Operators

5 % 2 =	1
5 % -2 =	1
-5 % 2 =	-1
-5 % -2 =	-1

```
int x=30600;
int y=15236;
int z=x*y/x;
```

Unexpected results

1.8.3 Relational (Comparison) Operators

- The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

- Note: There is no boolean data type in c; 0 means false and non-zero means true.

1.8.4 Bitwise (Logical) Operators

- Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for **&**, |, and ^ is as follows:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

1.8.4 Bitwise (Logical) Operators

- The following table lists the bitwise operators supported by C. Assume variable 'A' holds 7 and variable 'B' holds 5, then

Operator	Description	Example
&	**Binary AND Operator** copies a bit to the result if it exists in both operands.	(A & B) = 5, i.e., 0000 0101
|	**Binary OR Operator** copies a bit if it exists in either operand.	(A | B) = 7, i.e., 0000 0111
^	**Binary XOR Operator** copies the bit if it is set in one operand but not both.	(A ^ B) = 2, i.e., 0000 0010

1.8.5 Short-Circuit Operators

- Following table shows all the logical Short-Circuit operators supported by C language. Assume variable A holds 1 and variable B holds 0, then

Operator	Description	Example
&&	Short-Circuit AND Operator. If left operand equal false the result will be false without check on the right operand.	(A && B) is false= 0.
||	Short-Circuit OR Operator. If left operand equal true the result will be true without check on the right operand.	(A || B) is true= 1.

1.8.6 Shift Operators

- The following table lists the bitwise Shift operators supported by C. Assume variable 'A' holds 60, then:

Operator	Description	Example
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

1.8.7 Assignment Operators

- The following table lists the assignment operators supported by the C language:

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A

1.8.7 Assignment Operators

- The following table lists the assignment operators supported by the C language:

Operator	Description	Example
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

1.8.8 Misc Operators

• There are a few other important operators including **sizeof** and ***ternary*** supported by the C Language.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is float, will return 4.
&	Returns the address of a variable.	&a; returns the logical address of the variable a.
*****	Pointer to a variable.	*a; where a is a pointer variable
? :	Ternary Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Condition(s) ? True statement : false statement;

Ex: z=(x>y) ? 10 : 0;

Ex: (x>y) ? Z=10 : z=0;

1.8.9 Operators Precedence in C

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right

1.8.9 Operators Precedence in C

Category	Operator	Associativity
Logical Short-Circuit AND	&&	Left to right
Logical Short-Circuit OR	||	Left to right
Ternary	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Assignments

- Install the C environment
- Write a C program to test the different format specifiers with “printf “
- Write a C program to read a character from the user and print it and its ASCII code.
- Write a C program to display the octal and the hexadecimal representation of an integer number.

Control Statements

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

Content

- Control Statements in C
  - Branching Statements
  - Looping Statements
- Break Statement
- Continue Statement
- Comments in C
- Introduction to Magic Box Assignment

2.1 Control Statements

- There are some statements in all of programming languages to control the flow of the program execution; like conditional statements or repeating statements.
- In C there are two categories of control statements:
  - Branching Statements: which has two statements:
    - **if** statement
    - **switch** statement
  - Looping Statements: which has three statements:
    - **for** statement
    - **while** statement
    - **do .. while** statement

2.1.1 Branching Statements

- IF statement form is:

```
if( Condition(s) )
{
        …
        …    //true statements
        …
}
[else]
{
        …
        …    //false statements
        …
}
```

Example:

```
int grade = 48;

if(grade > 60)
      printf("Pass");
else
{
      printf("Fail");
}
```

2.1.1 Branching Statements

- Switch statement form is:

```
switch(myVariable){
      case value1:
        …
        …
        break;
      case value2:
        …
        …
        break;
      default:
        …
}
```

- **int**
- char
- enum

2.1.1 Branching Statements

- Switch example:

```
int type;
scanf("%d",&type);
switch(type)
{   case 10:
        printf("Perfect");
        break;
    case 5:
    case 4:
        printf("below avarage");
        break;
    default:
        printf("Not accepted");
}
```

2.1.2 Looping Statements (Iteration)

- **For statement:** The for loop is used when the number of iterations is predetermined.
- Its syntax:

```
for (initial statement(s) ; continuous condition(s) ; repeated step(s))
{
        …
        …
        …
}
```

```
for (i=0 ; i<10 ; i++)
{
        printf("%d\t", i);
}
```

2.1.2 Looping Statements (Iteration)

- **<u>While Statement:</u>** The while loop is used when the termination condition occurs unexpectedly and is checked at the beginning.
- Its syntax:

```
while (condition(s))
{
      …
      …
      …
}
```

```
int x = 0;

while (x<10) {
    printf("%d\n", x);
    x++;
}
```

2.1.2 Looping Statements (Iteration)

- **Do .. While Statement:** The do..while loop is used when the termination condition occurs unexpectedly and is checked at the end.
- Its syntax:

```
do
{
        …
        …
        …

}
while(condition(s));
```

```
int x = 0;

do{
    printf("%d\n", x);
    x++;
} while (x<10);
```

2.2 Break Statement

- The break statement can be used in loops or switch.
- It transfers control to the first statement after the loop body or switch body.

```
......
while(age <= 65)
{
        ......
        balance = payment * 1;
        if (balance >= 25000)
                break;
}
......
```

2.3 Continue Statement

- The continue statement can be used Only in loops.
- Abandons the current loop iteration and jumps to the next loop iteration.

```
......
for( year=2000; year<= 2099; year++){
     if (year % 4 == 0)
           continue;
     printf(“Y = %d”, year)
}
......
```

2.4 Comments in C

- To comment a single line:

```
// write a comment here
```

- To comment multiple lines:

```
/*  comment line 1
    comment line 2
     comment line 3 */
```

2.5 Introduction to Magic Box assignment

- You have the following box and need to put the numbers from 1 to 9 in each cell without repeating and with another constraint: the summation of each row equals to 15 and the summation of each column equals to 15 and the summation of each diagonal equals to 15.

	0	1	2
0	**6**	**1**	**8**
1	**7**	**5**	**3**
2	**2**	**9**	**4**

2.5 Introduction to Magic Box assignment - Algorithm

- The main constraints:
  - The number of rows equal to the number of the columns.
  - The order of the box (N) must be odd number; 3X3 or 5X5 or 31X31 and so on.
  - The numbers to put in the box start from 1 to NXN
- The algorithm steps:
  - Put the number “1” in the middle of the first row
  - Repeat the following test for each number to decide the place of the next number starting from number “1” till number “NXN -1”:

```
If (CurrentNumber % N !=0 )
     { decrement the current row: with constraint to circulate if necessary
       decrement the current column: with constraint to circulate if necessary
     }
else
     { increment the current row: with constraint to circulate if necessary
       use the same column
     }
 go to the right place and put the next number
```

2.5.1 How to move the cursor in C?

- Some compilers has a function in ***conio.h*** library called ***gotoxy(),*** and some other has not.
- If your compiler has not this function you may make it in your program as follow:

```
void gotoxy(int x, int y)
{
    COORD coord;
    coord.X = x;
    coord.Y = y;
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
}
```

- You just write it before the main function for now.
- **Note:** you will learn how to write a function later.

Arrays and Strings

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Content

- Meaning of Arrays and why we need it: Application Level
- Array Characteristics
- How to declare an array in C: Abstract Level
  - How to access a certain element and how to store or retrieve data
- How the array is implemented in the memory: Implementation Level
- Multi-Dimensional Arrays
- String as a one-dimensional array of character
- String Manipulation
- Normal and Extended Keys

2.6 Meaning of the array and why do we need it – Application Level

- An array is a collection of items stored at ***contiguous*** memory locations and elements can be ***accessed randomly*** using ***indices*** of an array. They are used to ***store similar type of elements*** as in the data type must be the same for all elements. They can be used to store collection of primitive data types such as int, float, double, char, or long. And so, an array can store complex or derived data types such as the structures, another arrays; but from the same types and size.
- **Why do we need arrays?**
  - The idea of an array is to represent many instances of data in one variable while they have a logical relation among them while this relation can be mapped to indicator to any element of these instances.
  - For example: list of the grads in a subject for group of students in a class: number of student is an indicator, the grads are the data stored in the array elements

2.7 Array Characteristics

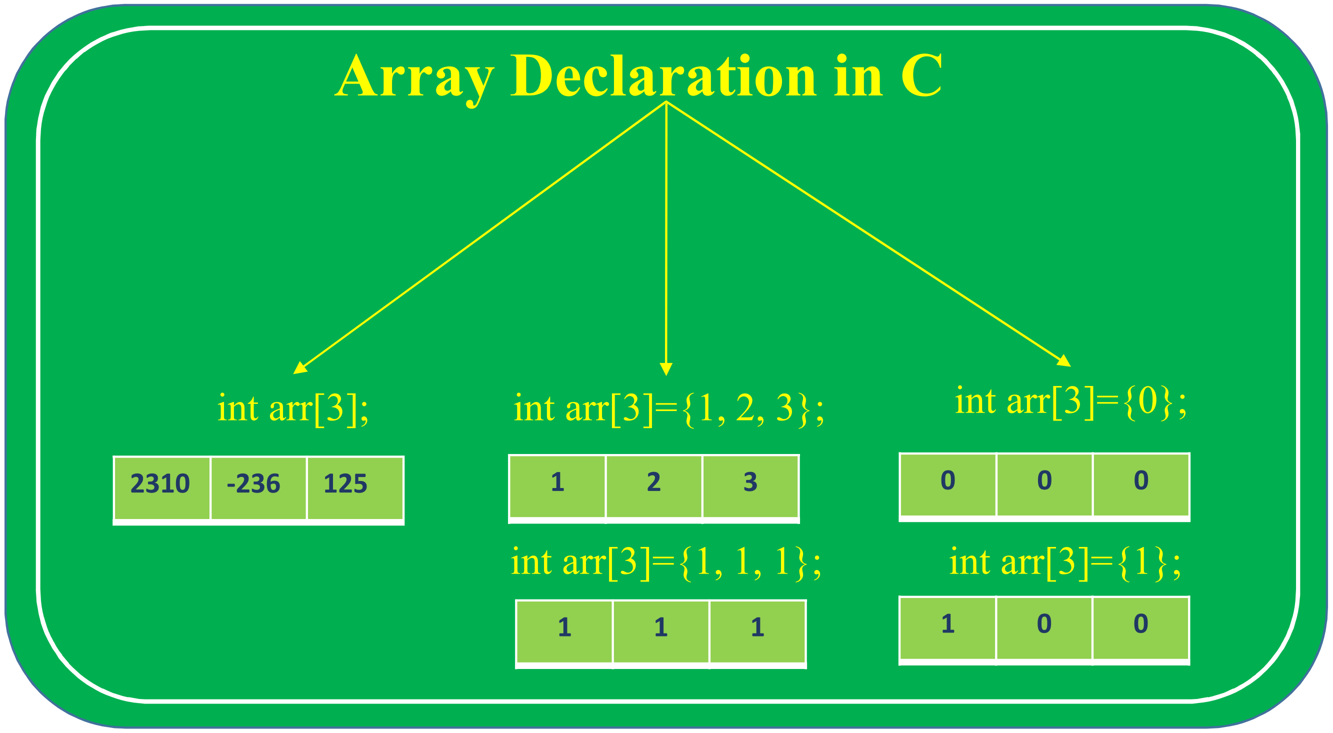
- There are seven characteristics for any array:

1. **Homogeneous**: All the elements data type must be the same
2. **Fixed size**: The size if the array can not changed at run-time (In C the size must be known in compilation-time).
3. **Contiguous**: All the instances of data for any array are allocated in contiguous memory locations.
4. **Indexed**: It uses the indexing technique to access any element (In C the index starts from 0 to (size -1))
5. **Ordered**: The places of the elements of any array are ordered.
6. **Finite (Limited)**: Any array has a first place and a last place. (Not circulated)
7. **Random or Direct Access**: The time consumed to reach to any element of an array is constant regardless its place

2.8 How to declare an array and access it in C: Abstract Level

- **Array declaration in C:**
  - There are various ways in which we can declare an array. It can be done by:
    - specifying its type and size,
    - by initializing it,
    - or both.
  - Array declaration by specifying size:
    - Data_Type Array_Name [Array_Size]; //some times we may use: Data_Type [Array_Size]Array_Name;
    - Ex: int arr [10];
  - Array declaration by initializing elements
    - Data_Type Array_Name [ ] = {val1, val2, val3, ….};
    - Ex: int arr [ ] = { 2, 4, -5, 88, -120}; // Compiler creates an array of size 5.
  - Array declaration by specifying size and initializing elements
    - Data_Type Array_Name [Array_Size] = {val1, val2, val3, ….};
    - Ex: int arr [10] = { 2, 4, -5, 88}; // Compiler creates an array of size 10, initializes first 4 elements as // specified by programmer and the rest 6 elements as 0.

2.8 How to declare an array and access it in C: Abstract Level



2.8 How to declare an array and access it in C: Abstract Level

- **Accessing Array Elements in C:**
  - Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1
  - Ex:

```
void main(void)
{       int arr[5];
        arr[0] = 5;
        arr[2] = -10;
        arr[3 / 2] = 2; // this is same as arr[1] = 2
        arr[3] = arr[0];
        printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);  // Output:  5   2   -10  5
}
```

- Note: There is no index out of bounds checking in C. ***Why?***

2.9 How the array is implemented in the memory: Implementation Level

- At first we need to clarify how the compiler declare and store the variables, by generate a ***Variables Vector***. It may look like the following table:
- Ex: int x; float f

Var Name	Type	Size	Address	…	…	…
x	int	2	2120			
f	float	4	3240			

- And so the compiler generate a vector for arrays, it may be look like the following table:
- Ex: int arr[10];

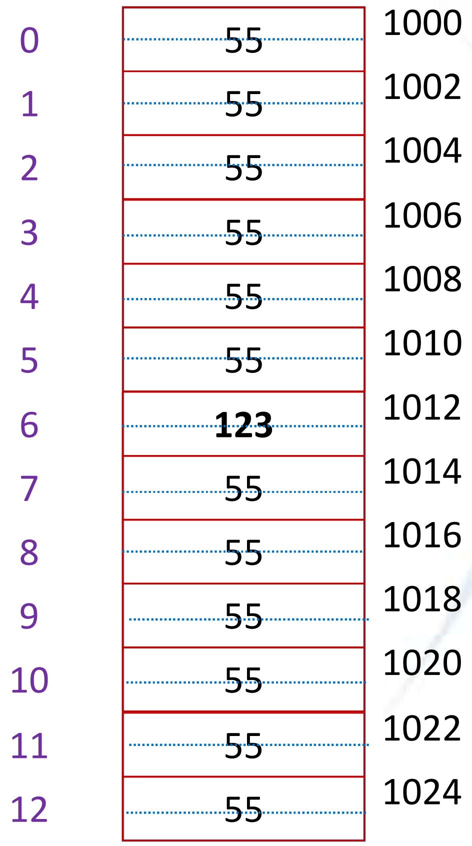
Array Name	Type	Size	Element Size	Base Address	…	…
arr	Int	10	2	1000		

2.9 How the array is implemented in the memory: Implementation Level

- The allocation of the array will be in memory starting from base address –as the address of first byte allocated to the array- with long of bytes equal to number of elements X element size. For the above example the base address is 1000 and the size of array in bytes = 10X2 =20 bytes; so the allocated locations for the array in memory from 1000 till 1019 – contiguous -
- The compiler generate an equation to access any element of the array using the index of the element. It may be look like:
  - *the address of element with index I = base address + I X element size*
- For the above example: the equation will be:
  - The address for element indexed with I = 1000 + I X 2
  - Ex the address of the fourth element in the array (Index = 3) arr[3] is 1006
    address of arr[3] = 1000 + 3 X 2 = 1006
- Note: for the way of array implementation there is no index out of bounds checking in C.

2.9 How the array is implemented in the memory: Implementation Level

```
int arr[13],x;
arr[6] = 123;
X = arr[12];   // x = 55;

The address computing:
    base + index * element size;
Case arr[6]:
    1000+6*2 = 1012
Case arr[12]
    1000+12*2= 1024
```



2.10 Dealing with arrays by looping statements

- Ex:

```
int i , arr[10];
for(i  = 0 ; i < 10 ; i ++)
  scanf("%d", & arr[i]);
for (i = 9; i >=0 ; i --)
  printf("%d", arr[i]);
```

2.11 Multi-Dimensional Arrays

- Array may be array of arrays, that is mean, to access an array element you have to use multi-indices: each level of index represent a dimensional level.
- Declaration and accessibility: as two dimensional
  - Data_Type Array_Name [Dim1_Size][Dim2_Size];
  - Ex: int arr2d [4] [5];
  - arr2d[0][0] = 22;   // first element
  - arr2d [3][4] = 99; // last element
- The Equation for the two-dimensional array:
  - *the address of element with index A , B = base address + A X 2nd Dim_array_size*
    + B X   data_element_size

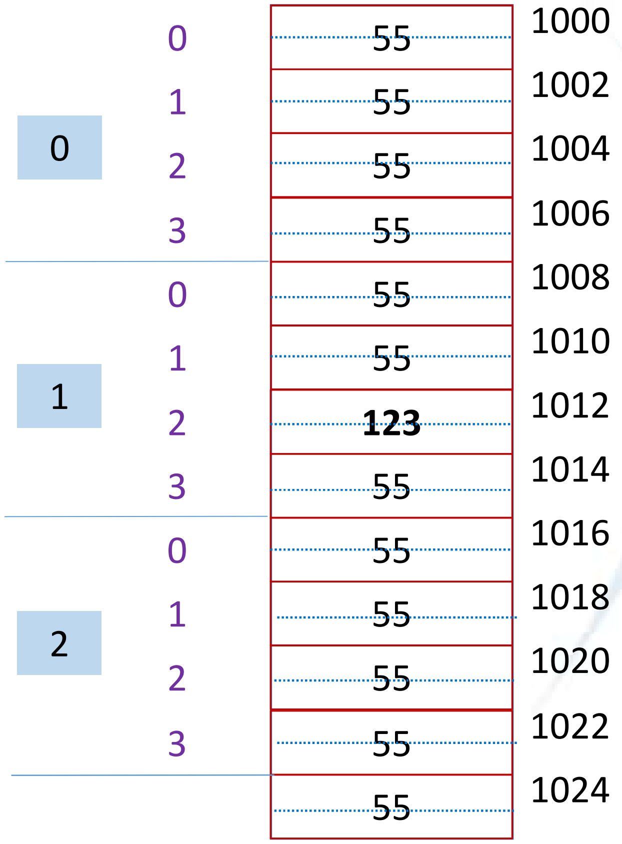
  Ex: address of arr2d[1][2] = base address + 1 X ($\underline{5 X 2}$) + 2 X 2

- Declaration of N dimension:
  - Data_Type Array_Name [Dim1_Size] [Dim2_Size] [Dim3_Size] …. [DimN_Size];

2.11 Multi-Dimensional Arrays

```
int arr2d[3][4],x ;
arr2d[1][2] = 123;
x = arr2d[2][3];
```

```
The address computing:
   base address + 1st index * 2nd dim array size
                + 2nd index * element size
Case arr2d[1][2]: 1000 + 1*(4*2) + 2*2 = 1012
Case arr2d[2][3]: 1000 + 2*(4*2) + 3*2 = 1022
```



2.12 String

- There is no string data type in C language, the dealing with a string in C is represented by using a one dimensional array of character.
- In C programming, a string is a sequence of characters terminated with a null character **‘\0’**
- There are two ways to declare a string in c language:
  - By char array
  - By string literal
- Let's see the example of declaring string by char array in C language:
  - char str[10]={'C', ‘-', 'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
- As we know, array index starts from 0, so it will be represented as in the figure given below:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
C	-	P	r	o	g	r	a	m	\0

2.12 String

- While declaring string, size is not mandatory. So we can write the above code as given below:
  - char str1[]={'C', ‘-', 'P', 'r', 'o', 'g', 'r', 'a', 'm', '\0'};
- We can also define the string by the string literal in C language. For example:
  - char str2[]=“C-Program”;
- In such case, '\0' will be appended at the end of the string by the compiler.
- There are two main differences between char array and literal:
  - We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
  - The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

2.12 String

- **Notes:**
- char str[] = {‘A’, ‘S’, ‘A’, ‘D’}; // is this a String or not?
- char str1 [5] = {‘A’,’L’,’Y’,’\0’}; // is this a String or not?
- char str2 [6] = “Information”; // is this a String or not?
- str2 = “Technology”; // Can we assign a string to array of char after declaration? **NO**

2.12 String Manipulation

- **<u>Read a string from a user:</u>**
  - By using **scanf("%s", str);** // problem with spaces
  - By using other functions for read lines like: **gets(str);** // accept the spaces
- **<u>Print a string on screen:</u>**
  - By using **printf("%s", str);**
  - By using other functions like: **puts(str);**
- **<u>Functions in string.h:</u>** there are many functions like:
  - **strlen(str)** : return the number of characters in the string – before the terminator
  - **strcat(str1,str2)** : concatenation the second string to the first string
  - **strcpy(destination, source)** : copy the source to the destination
  - **strcmp(str1, str2)** : compare between the two strings and return 0, or +ve value or –ve value !!
    - 0 : if str1 equals to str2
    - +ve: if str1> str2 : if the ASCII value of the first unmatched character is greater than the second and return the subtraction.
    - -ve if str1< str2 : if the ASCII value of the first unmatched character is less than the second and return the subtraction.

2.12 String Manipulation

- **<u>Functions in string.h:</u>** there are another functions compiler dependent like:
  - **strlwr()** : converts string to lowercase, found in *Borlandc* or *Turboc*
  - **strupr()** : converts string to uppercase, found in *Borlandc* or *Turboc*
  - **strrev(str)** : reverse characters of a string, found in *Borlandc* or *Turboc*
  - **strcmpi(str1, str2)** : compare between the two strings –case-insensitive- and return 0, or +ve value or –ve value !!, found in *Borlandc* or *Turboc*
    - 0 : if str1 equals to str2
    - +ve: if str1> str2 : if the ASCII value of the first unmatched character is greater than the second and return the subtraction.
    - -ve if str1< str2 : if the ASCII value of the first unmatched character is less than the second and return the subtraction.
  - **strstr(s1, s2)** : Find the first occurrence of a substring(s2) in another string (s1), and return the address of the occurrence if success or return null, found in *Borlandc* or *Turboc*.

2.13 Normal Keys and Extended Keys

- Each key in the keyboard has an ASCII code, which is limited by one byte.
- At the first of computer generation the keyboard was limited not like today; there was arrows or page up or down of function keys, …. etc.
- So, the new added keys needed ASCII code to be manipulated, so the Extended Keys are appeared.
- Its ASCII encapsulated in two bytes, the lowest byte equals null and the second byte has a code.
- When you pressed on an extended key there was 2 bytes are stored in the keyboard local buffer.
- The following program clarify how to now any key if it is normal or extended and print its code.

2.13 Normal and Extended Keys

```
Void main (void)
{
    char ch;
    flushall();
    printf("\n Press the key u want to know its type and code");
    ch=getch();
    if(ch!=NULL)//Some compilers not accept, replace with -32 or 224
        printf("\nIt is a normal key with code = %d",ch);
    else{
        ch=getch();
        printf("\nIt is a Extended key with 2nd byte code = %d",ch);
    }
    getch();
}
```

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Assignments

- Write a C program to implement the algorithm of the Magic Box puzzle.
- Write a C program to receive numbers from the user, and exit when the sum exceeds 100.
- Write a C program to print a simple menu with 3 choices, when select one choice print the choice word and exit.
- Write a C program to print the multiplication table in ascending order from table 1 to table 10 sequentially and separated by group of “ *’s ”.
- Rewrite the previous program to print them in descending order.
- Write a program to read an array and print it using 2 for loops?
- Write a program to find the maximum and minimum values of a set of numbers using a single dimension array.
- If you have a matrix of dimension 3*4. Write a program to read it from the user and find the sum of each row & the average of each column

Structures, array of structures and Modularity

Java™ Education and Technology Services

Invest In Yourself ,
Develop Your Career

Content

- Structure Meaning
- Structure definition (Abstract level)
- Structure declaration (Abstract level)
- Accessing Structure Members (Abstract level)
- Structure implementation in the memory (Implementation level)
- Array of Structures

3.1 Structure Meaning

- Structure is a user-defined datatype in C language which allows you to combine data of different types together. Structure helps to construct a complex data type which is more meaningful.
- **<u>For example:</u>** If I have to write a program to store Employee information, which will have Employee’s name, age, address, phone, salary etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.
- In structure, data is stored in form of records.

3.2 Structure definition (Abstract level)

- ***struct*** keyword is used to define a structure. ***struct*** defines a new data type which is a collection of primary and derived datatypes.

```
struct structure_Name
{
   type member1;
   type member2;
   type member3;
    ...
/* declare as many members as desired, but the entire
structure size must be known to the compiler. */
}[structure_variables];
```

3.2 Structure definition (Abstract level)

- Such a struct declaration may also appear in the context of a *typedef* declaration of a type alias or the declaration or definition of a variable:

```
typedef struct tag_name {
    type member1;
    type member2;
    type member3;
        ...
} struct_alias;
```

3.2 Structure definition (Abstract level)

- Example of Employee

```
struct Employee
{
        int ID;
        float Age;
        float salary
        float deduct;
        float bonus;
        char Name[51];
};
```

- To make the definition of a structure available to all functions in you program, you have to define it as part of preprocessing part in a C program.

3.3 Declaration a variable from a structure

- Structure variable declaration is similar to the declaration of any other datatype. Structure variables can be declared in following two ways:
  - Declaring structure variable separately,
  - Declaring Structure variables with structure definition
- **Declaring structure variable separately**

```
    struct Employee e1;
```

- **Declaring Structure variables with structure definition**

```
struct Employee
{
        int ID;
        float Age;
        float salary
        float deduct;
        float bonus;
        char Name[51];
} e2;
```

3.3 Declaration a variable from a structure

- **Initialization of a structure variable**

```
    struct Employee e1 = {213, 46, 3562.12,
                          324.2, 2.5,
                          "Mohamed Aly"};
```

3.4 Accessing Structure Members

- Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the structure variable using a dot "." operator also called *period* or *member access operator.* For Example:

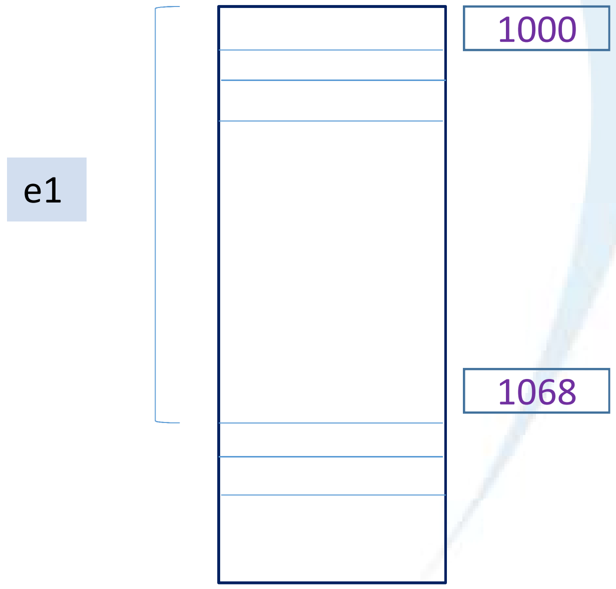
```
struct Employee e1;
e1.ID = 213;
scanf("%f",&e1.salary);
gets(e1.Name);
strcpy(e1.Name, "Hossam");
e1.Name[1] = 'M';
f = e1.salary + e1.bonus - e1.deduct;
```

3.5 Structure implementation in the memory (Implementation level)

- When the compiler read the definition of a structure it will create a structure table for that structure datatype, it looks like:

Struct Employee Table

Field Name	Field Type	Field Size	offset
ID	int	2	0
Age	float	4	2
Salary	float	4	6
deduct	float	4	10
Bonus	float	4	14
Name	char[51]	51	18
		69	



3.5 Structure implementation in the memory (Implementation level)

- Example:

```
e1.Salary = 333.3;
```

The offset of the field ‘Salary’ is added to the address if variable e1to reach to the address of the field and store the data.

3.6 Array of Structure

- The form is:

  struct Struct-Name Array_Name [Array_size];
- Example:

```
struct Employee empArr[5];

empArr[2].salary = 2315.6;
gets(empArr[3].Name);
scanf("%f",&empArr[0].bonus);
empArr[4].Name[2] = 'T';
```

- Note: if we use ***typedef*** keyword when defining the structure we can use the *alias name* directly without need to use the ***struct*** keyword

3.6 Array of Structure

- The array of structures initialization can be done by assign the values for each structure of the array, for example:

```
struct Employee empArr[2]= {
          {10, 35.2, 4500, 234.7, 200, “Hassan Aly”},
          {20, 42.3, 7500, 456.3, 450, “Mohsen Ayman”} };
```

Modularity In C Functions

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Content

- Modularity
- Function Header and prototype
- Function Parameters and Return Data Type
- Function Body
- Functions Arguments
- Sequence Passing arguments in C
- Recursion and its Constraints
- Introduction to Pointers
- Call by Value and Call by Address

3.8 Meaning of Modularity

- Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.
- In C the function is the tool to perform the modularity. Each function has its objectives to execute, it may be divided into many smaller functions to perform it.
- Functions can call any other functions, either the main function itself – with constraints for avoiding the infinite loop case-
- Functions may call itself, called recursive function with constraints too.

Function Header, Body, and Prototype

- The function consists of:
  - Header,
  - Body
- Function header consists of 3 parts as follow:

  Return_Datatype   Function_Name (List_OF_Input_Parameters)
- Function Body which contains the implementation of the algorithm to execute the function work which is written inside **{ }**
- We can write the function – header and body – before the function will call it, we can put it in any place in the file but after write its prototype in the preprocessing part before any functions.
- Function prototype is the copy of function header with "**;**" at the end of header

  Return_Datatype   Function_Name (List_OF_Input_Parameters);

3.8.1 Example 1

```
int sum2Var(int A, int B)
{
    int sum;
    sum = A+ B;
    return sum;
}
```

- The reserved word “***return***” are used with functions to terminate its work and return to the caller function, if the called function is expected to return datatype, the using of return must added with the value – or variable- with the return datatype.
- How this function is called and how to write its prototype?

3.8.1 Example 1

```
int sum2Var(int A, int B);        // Function prototype
void main (void)
{
     int x=5, y=7, z;
     z = sum2Var(x, y);           // Call the function with vars as arguments
     z = sum2Var(-7, 12);         // Call the function with values as args
     printf ("%d", sum2Var(51, -31)); //We can call a function as an argument
}
int sum2Var(int A, int B)                 // Function Header
{
     int sum;                             //
     sum = A+ B;                          //   Function Body
     return sum;                          //
}
```

3.9 The different types of functions

- Function may have no return datatype, and may have no list of input parameters; write ***void*** in both places.
- Function may have no return datatype, and may have list of input parameters; write ***void*** at the beginning of header and write the type and name for each input parameter.
- Function may have return datatype, and may have no list of input parameters; write the return data type in the header of function –either primitive or complex- and void in place of list of input parameters
- Function may have return datatype, and may have list of input parameters; write the return data type in the header of function –either primitive or complex- and write the type and name for each input parameter.

3.10 Parameters and Arguments

- The term parameter refers to any declaration within the parentheses following the function name in a function declaration or definition; the term argument refers to any expression within the parentheses of a function call.
- The following rules apply to parameters and arguments of C functions:
  - Except for functions with variable-length argument lists, the number of arguments in a function call must be the same as the number of parameters in the function definition. Or no parameters nor arguments
  - The maximum number of arguments (and corresponding parameters) is 253 for a single function.
  - Arguments are separated by commas. However, the comma is not an operator in this context
  - Arguments are passed by value; that is, when a function is called, the parameter receives a copy of the argument's value
  - The scope of function parameters is the function itself. Therefore, parameters of the same name in different functions are unrelated.

3.10.1 The sequence of passing the arguments to the function

- We pass different arguments into some functions. Now one questions may come in our mind, that what the order of evaluation of the function parameters. Is it left to right, or right to left?
- To check the evaluation order we will use a simple program. Here some parameters are passing. From the output we can find how they are evaluated.

Example 2:

```
#include<stdio.h>
void test_function(int x, int y, int z) {
   printf("The value of x: %d\n", x);
   printf("The value of y: %d\n", y);
   printf("The value of z: %d\n", z);
}
main() {
   int a = 10;
   test_function(a++, a++, a++);
}
```

Example 2:

```
Output will be:
The value of x: 12
The value of y: 11
The value of z: 10
```

- From this output we can easily understand the evaluation sequence. At first the z is taken, so it is holding 10, then y is taken, so it is 11, and finally x is taken. So the value is 12.

3.11 The main return values

- The return value for main indicates how the program exited. Normal exit is represented by a 0 return value from main. Abnormal exit is signaled by a non-zero return.
- As old fashion in C programming the values 1, 2, or 3 represent abnormal termination and represent the following state for each value:
  - 1 : abnormal exit without saying the reason
  - 2 : abnormal exit for memory problem (could not allocate memory, …)
  - 3 : abnormal exit for I/O problem (could not change driver mode, …)
- In general the using values are:
  - Return 0 for normal exit : like calling function exit(0)
  - Return non-zero value for abnormal exit: like calling function exit(1)

3.12 Recursion

- Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.
- The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.
- **What is the base condition in recursion?** In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

3.12 Recursion

- **How a particular problem is solved using recursion?** The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 1. We return 1 when n = 1.
- **Why Stack Overflow error occurs in recursion?** If the base case is not reached or not defined, then the stack overflow problem may arise.
- **How memory is allocated to different function calls in recursion?** When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

3.12.1 Recursion Example: Factorial

- Iteration version:

```
int Fact( int n)
{
    int result=1, i;
    for( i = n; i > 1; i--)
        result *= I;
    return result;
}
```

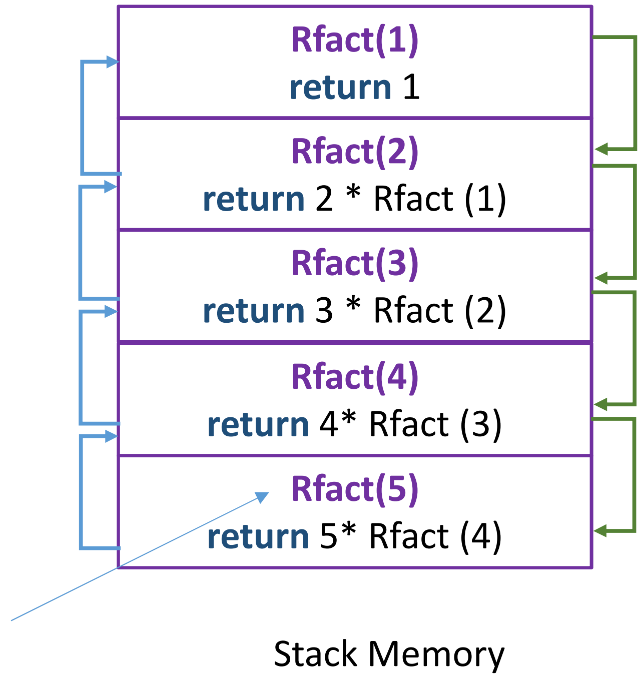
3.12.1 Recursion Example: Factorial

- Recursive version:

```
int RFact( int n)
{
    if (n == 1)
        return 1;
    return n * Rfact( n-1 );
}
```

3.12.2 How to trace a recursive function

```
RFact(5)= 5 *
     RFact(4) = 4 *
          Rfact(3) = 3 *
               RFact(2) = 2 *
                    RFact(1) = 1
```



3.12.3 Example 2: Fibonacci series

- The following example generates the Fibonacci series for a given number using a recursive function

```
#include <stdio.h>
int fib (int i) {

    if(i == 1 || i == 2) {
      return 1;
    }

    return fib (i-1) + fib (i-2);
}
```

3.12.3 Example 2: Fibonacci series

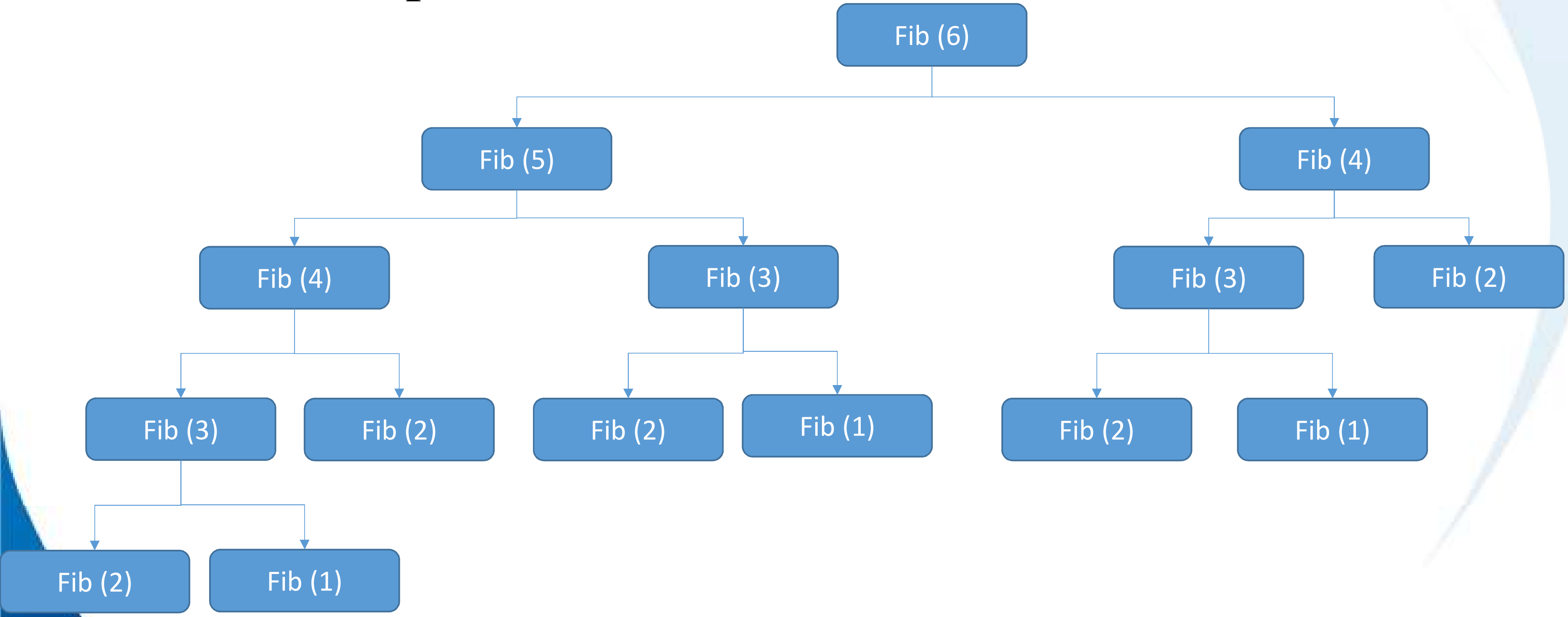
```
int  main(void) {

   int i;

   for (i = 0; i < 10; i++) {
      printf("%d\t\n", fib(i));
   }

   return 0;
}
```

3.12.3 Example 2: Fibonacci series

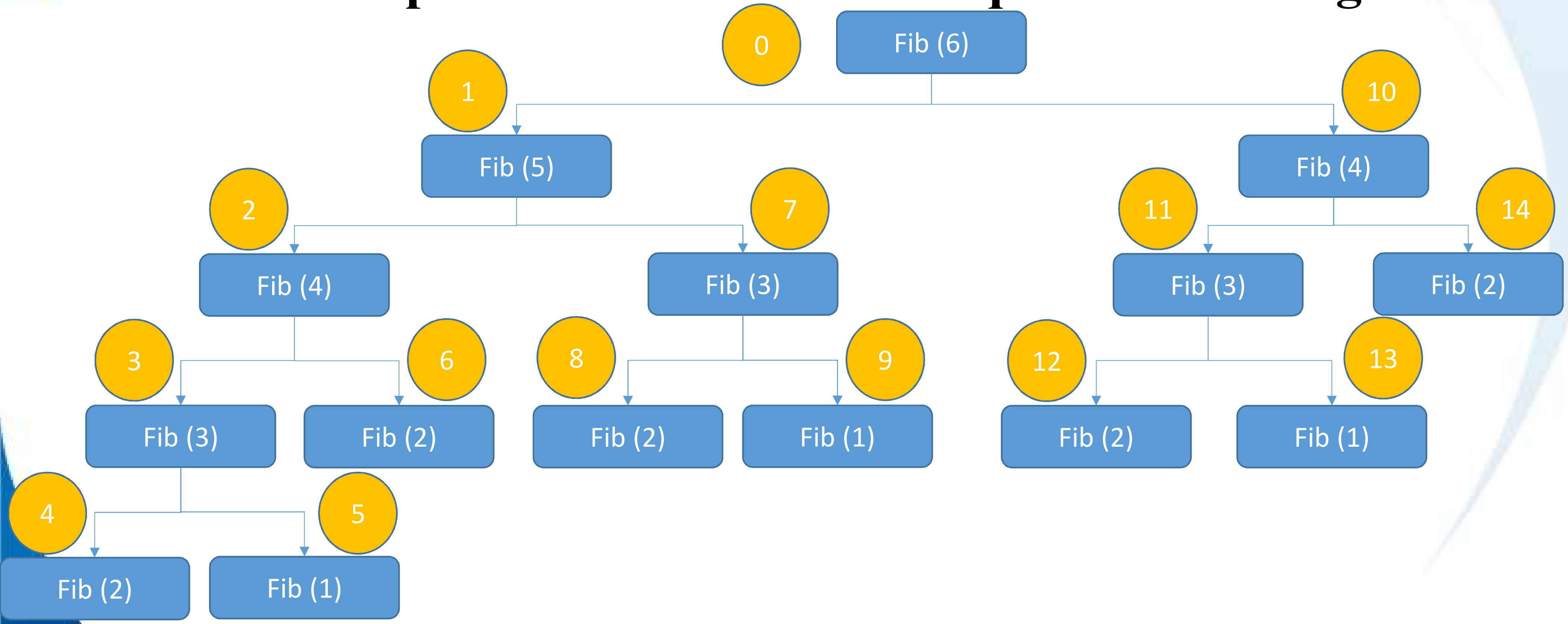
The output as follow:

```
1
1
2
3
5
8
13
21
34
55
```

3.12.3 Example 2: Fibonacci series



3.12.3 Example 2: Fibonacci series sequence of calling



3.12.4 Advantages and Disadvantages of Recursion

- Recursion makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.
- That being said, recursion is an important concept. It is frequently used in data structure and algorithms. For example, it is common to use recursion in problems such as tree traversal.
- There is an important technique used in algorithms dependents on recursion concept, ***Divide-and-conquer*** which is used in many types of problem solving such sorting and searching problems, such as Merge sort and binary search algorithms.

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Assignments

- Write a recursive function to compute the power operation: X^Y
- Write a program to receive one employee's data display the code, name, and net salary.
- Write two functions to read an Employee's data and other to print its Name with the net salary.
- Write a program to receive data into an array of 5 employees, then display the code, name, and net salary for each.

Pointer Datatype

Content

- Introduction to Pointers
- Pointers' Arithmetic
- Arrays and pointers
- Pointers Comparisons
- Using Pointers to call by address
- Using pointers to pass an array to function
- Introduction to dynamic allocation
- malloc(), realloc(), and free() functions
- Dynamic allocations of arrays
- Pointer to pointer and array of pointers
- The standard main header in c

4.1 Introduction to Pointers

- Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers.
- **What are Pointers?** A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.
- Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is:

```
    Datatype * Pointer_Name;
Ex: int * ptr;
```

- ***Datatype*** is the pointer's base type; it must be a valid C datatype and ***Pointer_Name*** is the name of the pointer variable. The asterisk **'*'** used to declare a pointer

4.1 Introduction to Pointers

- The actual data type of the value of all pointers, whether pointer to integer, float, character, or otherwise, is the same, ***an hexadecimal number that represents a memory address (Logical address)***. The only difference between pointers of different datatypes is the datatype of the variable or constant that the pointer *points to*.
- The key to deal with pointer is to know at any time if you deal with its contents or the contents of what it points to
- As you know, every variable has memory location (s) and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined:

4.1 Introduction to Pointers

```
#include <stdio.h>
int main () {
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );
   return 0;
}
```

- The output:

```
Address of var1 variable: fff4
Address of var2 variable: ffea
```

4.1.1 How to Use Pointers?

- There are a few important operations, which we will do with the help of pointers very frequently.
  - We define a pointer variable,
  - Assign the address of a variable to a pointer and,
  - Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.
- The following example makes use of these operations:

4.1.1 How to Use Pointers?

```
#include <stdio.h>
void main (void) {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var;  /* store address of var in pointer variable*/
   printf("Address of var variable: %x\n", &var  );
                /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );
                /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
}
```

4.1.1 How to Use Pointers?

- The output of the above example:

```
Address of  var  variable: fff4
Address stored in  ip  variable: fff4
Value of  *ip  variable: 20
```

4.2 Pointers Arithmetic:

- A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value.
- There are four arithmetic operators that can be used on pointers: **++, --, +,** and **–**
- To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address **1000**. Assuming 16-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr ++;
```

- After the above operation, the **ptr** will point to the location **1002** because each time **ptr** is incremented, it will point to the next integer location which is 2 bytes next to the current location.
- This operation will move the pointer to the next memory location without impacting the actual value at the memory location.
- If **ptr** points to a character whose address is **1000**, then the above operation will point to the location **1001** because the next character will be available at **1001**.

4.2 Pointers Arithmetic: Example

- Consider the following piece of code:

```
    int x = 5;       // consider address of x is 1000
    int * ptr = &x; // the ptr is points to address 1000
```

- What will happen if we run each of the following statements in x an ptr?

```
  y = * ptr ++;    // ptr=1002, x=5, y= 5 (compiler dependent)
  y = * (ptr)++;   // ptr=1002, x=5, y= 5 (compiler dependent)
  y = (*ptr)++;    // ptr=1000, x=6, y=5
```

4.2 Pointers Arithmetic: Example

```
int main(void) {
   char arr[] = “ITI World";
   char *ptr = &arr[0];
   char *ptr1 = &arr[0];
   char *ptr2 = &arr[0];
   ++*ptr; // increment the content of the position pointed by ptr
   printf("\n value of *ptr = %c \t %x",*ptr, ptr);
   *ptr1++;// get the content of the position after increment ptr1
   printf("\n value of *ptr1= %c \t %x", *ptr1, ptr1);
   *++ptr2;// get the content of the position after increment ptr2
   printf("\n value of *ptr2 = %c \t %x", *ptr2, ptr2);
   return 0;
}
```

4.2 Pointers Arithmetic: Example

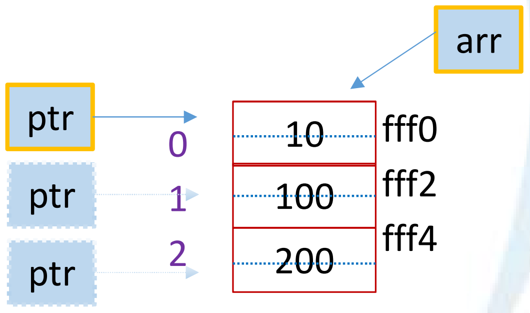
```
Output as follow:

value of *ptr = J  ffe8
value of *ptr1= T  ffe9
value of *ptr2= T  ffe9
```

4.3 Arrays and Pointers

- The array name is the base address; the first address dedicated to the array elements, so we can assign the array name to a pointer to the same datatype of array.
- The variable pointer can be incremented, unlike the array name which cannot be incremented because it is a ***constant pointer***.
- The following program increments the variable pointer to access each succeeding element of the array

4.3 Arrays and Pointers

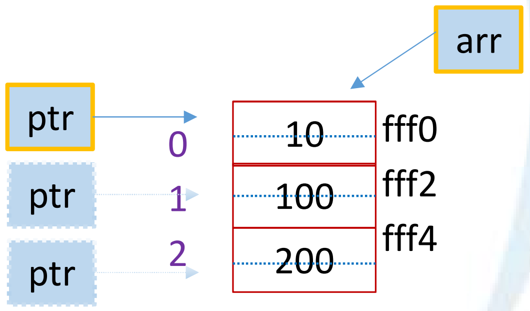
```
void main (void) {
   int  arr[] = {10, 100, 200};
   int  i, *ptr;
   /* let us have array address in pointer */
   ptr = arr;
   for ( i = 0; i < 3; i++) {
      printf("Address of arr[%d] = %x\n", i, ptr );
      printf("Value of arr[%d] = %d\n", i, *ptr );
      /* move to the next location */
      ptr++;
   }
}
```



4.3 Arrays and Pointers

- The output will be as follow:

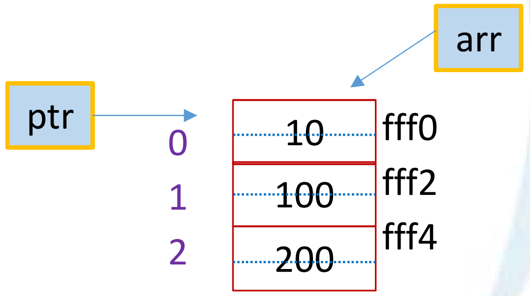
```
Address of arr[0] = fff0
Value of arr[0] = 10
Address of arr[1] = fff2
Value of arr[1] = 100
Address of arr[2] = fff4
Value of arr[2] = 200
```



4.3 Arrays and Pointers

- One of the advantages of using the pointers is: we can deal with locations in memory relatively to what pointer points to. For example:

```
void main (void) {
   int  arr[] = {10, 100, 200};
   int  i, *ptr;
   /* let us have array address in pointer */
   ptr = arr;
   for ( i = 0; i < 3; i++) {
      printf("Address of arr[%d] = %x\n", i, ptr+i );
      /* deal with the ith location after pointer points to*/
      printf("Value of arr[%d] = %d\n", i, *(ptr+i) );
   }
}
```



4.3 Arrays and Pointers

- We can deal with the name of the array as a pointer – but without trying to change it- i.e. we may use the pointer operator with array name and vise versa, we can use the array operator with the pointer. For example:

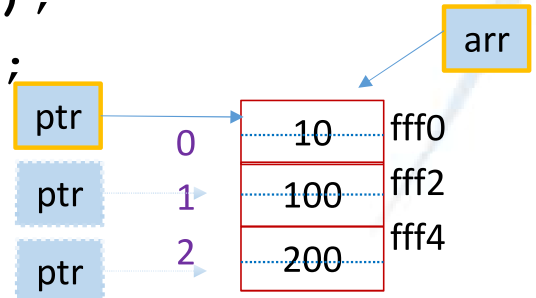
```
void main (void) {
   int  arr[10], i, *ptr;
   ptr = arr;
   for ( i = 0; i < 10; i++)
        scanf(“%d”, arr+i );
   for ( i = 0; i < 10; i++) {
        printf("Value of arr[%d] = %d\n", i, ptr[i] );
   }
}
```

4.4 Pointer Comparisons

- Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.
- The following program modifies the previous example − one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is *&arr[2]*

4.4 Pointer Comparisons

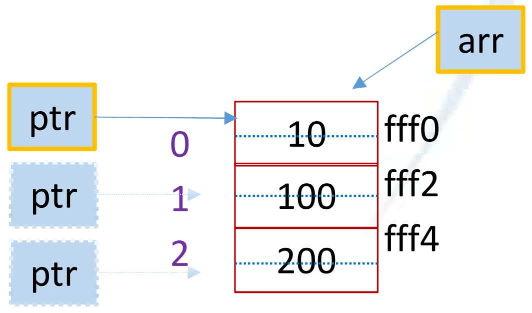
```
#include <stdio.h>
void main (void) {
   int  arr[] = {10, 100, 200, 400, 500};
   int  i=0, *ptr;
   /* let us have address of the first element in pointer */
   ptr = arr;
   while ( ptr <= &arr[2] ) {
      printf("Address of arr[%d] = %x\n", i, ptr );
      printf("Value of arr[%d] = %d\n", i, *ptr );
      /* point to the next location */
      ptr++;
      i++;
   }
```



4.4 Pointer Comparisons

- The output will be as follow:

```
Address of arr[0] = fff0
Value of arr[0] = 10
Address of arr[1] = fff2
Value of arr[1] = 100
Address of arr[2] = fff4
Value of arr[2] = 200
```



4.5 NULL Value in pointers

- If the pointer has the NULL value, that is mean it is not pointing to any location in the memory; i.e. it is not allowable to deal with what it points to.
- If you try to deal with what pointer points to with NULL value, the program is terminated immediately with run-time exception called ***"Null Pointer Exception"*** .
- Ex:

```
    int * ptr = NULL;
    * ptr = 5;  // causes Null Pointer Exception
```

- It is important to check on the value of the pointer content if it is not NULL before trying to deal with what it points

4.6 Using Pointers to call by address: Passing pointers to functions in C

- C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.
- Lets try to make a function to swap the values between two integers, lets try to thing make it call by value and discuss.

```
void swap (int A, int B)
{   int temp;
    temp = A;
    A = B;
    B = temp;
}
```

- What will happen when we call it with two variables are passed to it, see the following program.

4.6 Using Pointers to call by address: Passing pointers to functions in C

```
void main (void)
{   int x=5, y=7;
    printf("\nValues: x=%d, y=%",x,y);
    /* call swab function */
    swap(x, y);
    printf("\n Values after swapping: x=%d, y=%",x,y);
}
```

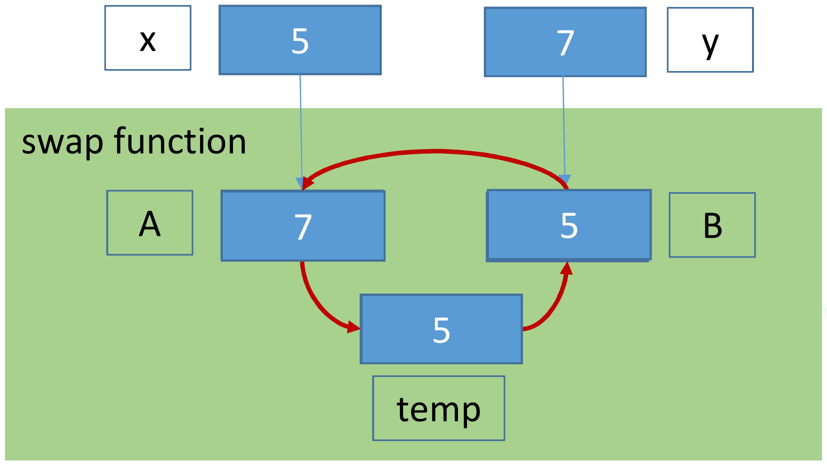
- The output will be:

```
Values: x=5, y=7
Values after swapping: x=5, y=7
```

4.6 Using Pointers to call by address: Passing pointers to functions in C

- Why this function not working? Actually it is working and the swapping is done, but between the copy of the variables not between the original variables, because we call the function *by value*, How?
- When we pass a variable as an argument to a function, a copy of the value of the variable is put in the input parameter; in different place as local variable for this function, so the working of the function execute on the local variable (a copy) not on the original variable

Function Call : `swap ( x , y)`



4.6 Using Pointers to call by address: Passing pointers to functions in C

- The solving of the above problem by using pointer to call the function and pass the address of the variables as arguments not the values of the variables; *Call by address*. As the following function: swap2

```
void swap2 (int * A, int * B)
{   int temp;
    temp = *A;
    *A = *B;
    *B = temp;
}
```

- The using of this function will be as follow:

4.6 Using Pointers to call by address: Passing pointers to functions in C

```
void main (void)
{   int x=5, y=7;
    printf("\nValues: x=%d, y=%",x,y);
    /* call swab function */
    swap2( &x, &y);
    printf("\n Values after swapping: x=%d, y=%",x,y);
}
```

- The output will be:

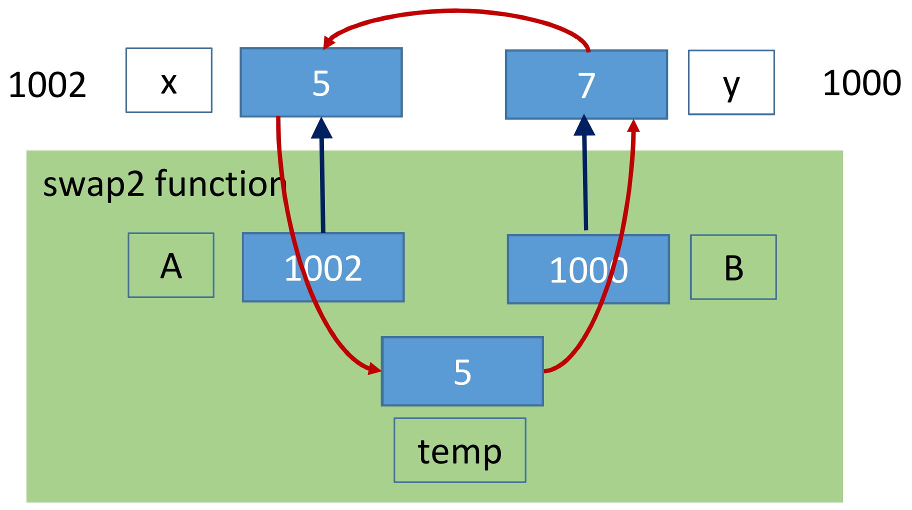
```
Values: x=5, y=7
Values after swapping: x=7, y=5   //Success result
```

- How it is working? Let’s see ….

4.6 Using Pointers to call by address: Passing pointers to functions in C

- The following simulation will explain how it is working

```
void swap2 (int * A, int * B)
{     int temp;
      temp = *A;
      *A = *B;
      *B = temp;
}
```



Function Call : `swap2 ( &x , &y)`

4.7 Using pointers to pass an array to function

- As we show early in the lecture, the array name is a fixed or constant pointer to the first byte dedicated to the array, so we can pass the array name to a function which has an input parameter as pointer. For example we have the following function:

```
void printArray(int * ptr, int size)
{ int i;
     printf("\n The values of the array:");
    for (i=0; i<size; i++)
        printf("\n element No %d = %d",i+1, ptr[i]);
}
```

- You may call this function as follow:

```
    int arr[25]={2,4,6,8,10,12};
    printArr(arr, 25);  // or u may use less than size
```

4.7 Using pointers to pass an array to function

• Ex: Make a function to count the length of a string, like strlen().

```
int stringLength(char * str)
{
    int count=0;
    while(str[count]!='\0')
        count++;
    return count;
}
```

4.7 Using pointers to pass an array to function

- Ex: using the abov function

```
void main(void)
{
    char name[]={“Hassan Aly Mohamed”}; // 18 character
    printf(“The lenth = %d” , stringLength(name));

}
```

4.8 Return a pointer from a function

- You may return a pointer datatype from a function if you want to return an address for a data but take in consideration this data – which you want to return its address – must be exist in memory after function return, i.e. if you return an address of a local data type it must be declared as ***static local variable*** (created at the first call to the function in heap section not in stack section and it will be exist until the program is terminated)
- Ex: A function to search a data in array and return its address:

4.8 Return a pointer from a function

```
int * searchAnInt(int * ptr, int size, int data)
{
    int i;
    for (i=0 ; i<size ; i++)
        if( ptr[i] == data )
            return ptr+i;

    return NULL;
}
```

4.8 Return a pointer from a function

```
void main (void)
{   int * add;
    int n, arr[10]={22, 3, 5, 88, -12, 6, 99, -3, 25, 169};
    printf("\n Enter a number to search it");
    scanf("%d",&n);
    add = searchAnInt(arr, 10, n);
    if(add==NULL) printf("\n The data u entered not exist");
    else printf("\n dat is exist with add: %x", add);
}
```

4.9 Pointer to a structure

- We have already learned that a pointer is a variable which points to the address of another variable of any data type like int, char, float etc. Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable. Here is how we can declare a pointer to a structure variable.

```
struct Employee e;
struct Employee * sptr;
sptr = &e;
```

4.9.1 Accessing structure members using Pointer

- There are two ways of accessing members of structure using pointer:
  - Using indirection (*) operator and dot (.) operator.
  - Using arrow (->) operator or membership operator.
- **Using Indirection (*) Operator and Dot (.) Operator**

```
    (*sptr).ID =  23;
    gets((*sptr).Name);
```

- **Using arrow operator (->)**

```
    sptr -> ID = 23;
    gets( sptr -> Name);
```

4.10 Dynamic Allocation

- One of the important using of pointers to make the dynamic allocation; allocate some of memory bytes at the runtime and deal with it. What is the dynamic allocation?
- As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.
- Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as *dynamic memory allocation* in C programming.
- To allocate memory dynamically, library functions are *malloc()*, *calloc()*, *realloc()* and *free()* are used. These functions are defined in the <stdlib.h> header file and <alloc.h>.

4.10.1 malloc() Function

- The name "malloc" stands for memory allocation.
- The ***malloc()*** function reserves a block of memory of the specified number of bytes. And, it returns a ***<u>pointer to void</u>*** which can be ***<u>casted</u>*** into pointers of any form.
- Syntax of malloc()

```
ptr = (castType*) malloc(size in bytes);
```

- Ex:

```
ptr = (int*) malloc(100); // allocate 100 bytes
fptr = (float*) malloc(50 * sizeof(float));
```

- The second statement allocates 200 bytes of memory. It's because the size of float is 4 bytes. And, the pointer *fptr* holds the address of the first byte in the allocated memory.
- The expression results in a **NULL** pointer if the memory cannot be allocated.

4.10.2 calloc() Function

- The name "calloc" stands for contiguous allocation.
- The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to **zero**.
- Syntax of calloc():

```
ptr = (castType*)calloc(repetation_No, size);
```

- Ex:

```
fptr = (float*) calloc(25, sizeof(float));
```

- The above statement allocates contiguous space in memory for 25 elements of type float and all of it are initialized by 0.

4.10.3 free() Function

- Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.
- Syntax of free():

```
    free(ptr);
```

- This statement frees the space allocated in the memory pointed by ptr.

4.10.4 realloc() Function

- The name “realloc" stands for reallocation
- If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.
- Syntax of realloc():

```
    ptr = realloc(ptr, New_Size);
```

- Here, ptr is reallocated with a New_Size bytes.

4.10.5 Examples on Dynamic Memory Alocation

```
void main(void)
{
    int * ptr, N;
    scanf("%d",&N);
    ptr= (int *)malloc(N*sizeof(int));
    for(i=0; ptr!=NULL&&i<N; i++)
        scanf("%d", &ptr[i]);
    for(i=0; ptr!=NULL&&i<N; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    free(ptr);
}
```

4.10.5 Examples on Dynamic Memory Alocation

```
void main(void)
{
    int * ptr, N;
    scanf("%d",&N);
    ptr= (int *)calloc(N, sizeof(int));
    for(i=0; ptr!=NULL&&i<N; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    for(i=0; ptr!=NULL&&i<N; i++)
        scanf("%d", &ptr[i]);
    for(i=0; ptr!=NULL&&i<N; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    free(ptr);
}
```

4.10.5 Examples on Dynamic Memory Alocation

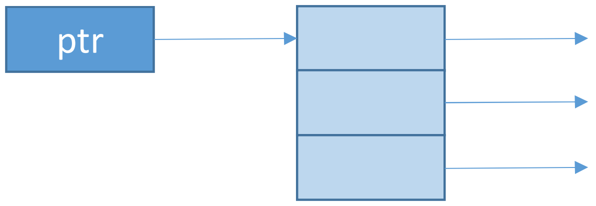
```
void main(void)
{   int * ptr, N, i;
    scanf("%d",&N);
    ptr= (int *)malloc(N*sizeof(int));
    for(i=0; ptr!=NULL&&i<N; i++)
        scanf("%d", &ptr[i]);
    for(i=0; ptr!=NULL&&i<N ; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    ptr= (int *)realloc(ptr,(N+5)*sizeof(int));
    for(i=0; ptr!=NULL&&i<(N+5) ; i++)
        printf("\n value ptr[%d]=%d", i, ptr[i]);
    free(ptr);
}
```

4.11 Array of Pointers

- There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available.
- Following is the declaration of an array of pointers to an integer :

```
int * ptr[3];
```

- It declares ptr as an array of 3 integer pointers. Thus, each element in ptr, holds a pointer to an int value.

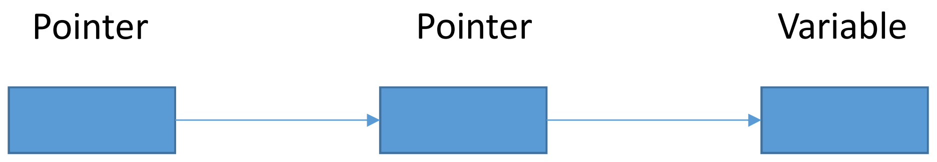


4.11.1 Example on array of pointers

```
void main(void)
{   char * strs[3];
    int i;
    for(i=0; i<3; i++)
        strs[i] = (char *)malloc(11 * sizeof(char));
    gets(strs[1]);
    strcpy(strs[0], “Hello”);
    strcpy(strs[2], “In ITI”);
    for(i=0; i<3; i++) printf(“%s ”,strs[i]);
}
```

4.12 Pointer to Pointer

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



4.12.1 Examples

```
void main (void) {
   int  var = 3000;
   int  *ptr;
   int  **pptr;
        /* take the address of var */
   ptr = &var;
        /* take the address of ptr using address of operator &
*/
   pptr = &ptr;
        /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
}
```

4.12.1 Examples

```
void main (void) {
   int i, j, N, M, **pptr;
   scanf(“%d”&N);         scanf(“%d”&M);
   pptr = (int **) malloc (N * sizeof(int *));
   for(i=0; i<N; i++)
        pptr[i] = (int *) malloc(M * sizeof(int));
   for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            scanf(“%d”, &pptr[i][j]);
 for(i=0; i<N; i++)
    for(j=0; j<M; j++)
        printf("Value of pptr[%d][%d] = %d\n",i,j, pptr[i][j]);
}
```

4.13 The standard header of the main function in C

```
int main (int argc, char ** argv)
```

- The main may return a value to the operating system to indicate the termination of the program done normally or there is an abnormal situation; 0 represent normal termination, other values means abnormal termination
- The operating system may pass arguments to the main, so the ***number of arguments*** is the first argument so it is an integer, and the ***arguments values*** is the second parameter as an array of strings – two dimensional array of character- so it is pointer to pointer to character.
- The first argument always is the name of the program you are running.

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself , Develop Your Career

Assignments

- Write a swap function using call by address.
- Using a function to read an array from user, and another one to print it.
- Write a function to make a string copy from source string to destination string. (like strcpy())
- Write a program to make an array of Employees with size determined at run time from a user and read its data and print the ID and net salary for each employee.
- Write a program to make an array of strings which number of strings is read from the user and so the size of each string is read from the user, after that read all strings and print it.

Miscellaneous topics in C

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

Content

- Static local variables
- Modifiers and access specifiers
- Enumerate datatype
- What is void data type in C
- Union datatype

Static Local Variable

- It is a variable declared inside a function with modifier ***static***, which make the variable to be allocated in the heap section instead of stack section.
- The life time of the static local variable is start at the first call of the function after the program execute, and end with the termination of the program.
- The scope of it inside the owner function, i.e. it is not accessible outside the function, except if and only if you got its address and then you may access it outside its scope.
- After first call of a function includes a static variable, the static local variable is already exist in memory, so when you recall the function again, the static local variable will not be allocated again and it includes the last value from previous call.
- The following simple example illustrates a static local variable work as a counter for the number of calling the function and return that counting after each calling.

Static Local Variable

```
int doSomeThing(void)
{   static int count = 1;
    …
    return count;
}
void main(void){
    int i=0;
    for( ; i<10; i++)
        printf("\n%d", doSomeThing());
}
```

Modifiers and Qualifiers in C Language

- Modifiers are keywords in c which changes the meaning of basic data type in c.
- Modifier specifies the amount of memory space to be allocated for a variable.
- Modifiers are prefixed with basic data types to modify the memory allocated for a variable.
- Qualifiers are keywords which are used to modify the properties of a variable are called type qualifiers.

Modifiers and Qualifiers in C Language

- Types of modifiers and qualifiers can categorized as follow:
  - Signing: **signed** and **unsigned**; all primitive data types are signed by default except character, it is unsigned by default
  - Sizing: **short**, **long;** the size of integer datatype is dependent on the platform work on it (2 or 4 bytes) is the short size so the long will be 4 or 8 bytes.
  - Variable modifiers: **auto**, **extern**, **static**; auto means auto allocated and auto deallocated; local variables,

  extern means the variable (or the function) will find else where; i.e. in another file, and static used to make static local variable, if it is used with global variables, that means its scope within the file include it only.
  - Pointers Modifiers: **near**, **far**, **huge**; to represent the level of accessibility of the pointer: within the memory allocated to the program or out – like physical address- huge like far but more accurate.
  - Qualifiers: **volatile**, **const;** volatile variable may be accessed by background routine.
  - Interruption: **interrupt**; used to represent a function called as interrupt service routine or part of it.

Enumeration Datatype in C

- **What is *enum* data type in C**
- Enumeration Types are a way of creating your own Type in C.
- It is a user-defined data type consists of integral constants and each constant is given a name.
- The keyword used for an enumerated type is ***enum***.
- The enumerated types can be used like any other data type in a program.
- Here is the syntax of declaring an ***enum***

```
    enum identifier{ value1, value2,...,valueN };
enum WeekDays{ Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

- Now any variable of enum days can take any one of the seven values.

```
    enum WeekDays holiday = Friday;
```

- Here, *holiday* is a variable of data type ***enum WeekDays*** and is initialized with value Friday.

What is void data type in C?

- The **void data type** is an empty data type that refers to an object that does not have a value of any type. Here are the common uses of **void data type**. When it is used as a function return type.

```
void myFunction(int i);
```

  - Void return type specifies that the function does not return a value.
- When it is used as a function's parameter list:

```
int myFunction(void);
```

  - Void parameter specifies that the function takes no parameters.

- When it is used in the declaration of a pointer variable:

```
void *ptr;
```

  - It specifies that the pointer is "universal" and it can point to anything. When we want to access data pointed by a void pointer, first we have to type cast it.

What is Union Datatype in C?

- A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.
- **Defining a Union**
- To define a union, you must use the ***union*** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for yur program. The format of the union statement is as follows:

```
union [union tag] {
   member definition;
      ...
   member definition;
} [one or more union variables];
```

- each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional.

The size of Union Datatype

- The following example illustrate that:

```
union Data {
   int i;
   float f;
   char str[20];
};
void main(void) {
   union Data data;
   printf("Memory size occupied by data:%d\n", sizeof(data));
}
```

- The output: `Memory size occupied by data : 20`

Accessing Union Members

- To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type. The following example shows how to use unions in a program using the above union defined.

Accessing Union Members

```
void main(void) {
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
}
```

- The ouput:

```
data.i : 19178
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Accessing Union Members

- Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.
- Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions

Accessing Union Members

```
void main(void) {
   union Data data;
   data.i = 10;
   printf( "data.i : %d\n", data.i);
   data.f = 220.5;
   printf( "data.f : %f\n", data.f);
   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);
}
```

- The output:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Object-Oriented Concepts and Terminologies

Content

- Introduction to Object-Oriented programing
- Object-Oriented Terminologies
  - Class, Object, Instance
  - Members:
    - Attributes (Data),
    - Behaviors (Functions or Methods)
  - Messages
  - Encapsulation: private, public
  - Abstraction
  - Polymorphism:
    - Overloading:
      - Function overloading
      - Operator overloading
    - Overriding

Content

- Inheritance (is-a relationship):
  - Multi-Level Inheritance
  - Multiple Inheritance
- Constructors and destructor
- Static members and Instance (non-static) members
- Virtual function and dynamic binding
- Abstract methods (Pure virtual function), Abstract class, and Concrete class
- Association
- Strong (Composition)
- Weak (Aggregation)

- Object-Oriented programming characteristics
- Advantages of Object-Oriented programing
- Complex Example (with class keyword)
- Constructors and destructor

5.1 Introduction to Object-Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields known as *attributes* or *properties*, and code, in the form of function known as *methods*.
- Object-oriented programming (OOP) is a programming paradigm based upon objects (having both *data* and *methods*) that aims to incorporate the advantages of *modularity* and *reusability*. Objects, which are usually instances of *classes*, are used to interact with one another to design applications and computer programs.
- Object Oriented programming (OOP) is a programming paradigm that relies on the concept of *classes* and *objects*. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.

5.1 Introduction to Object-Oriented Programming

- OOP has become a fundamental part of software development. Thanks to the ubiquity of languages like Java and C++, you can't develop software for *mobile* unless you understand the object-oriented approach. The same goes for serious *web* development, given the popularity of OOP languages like Python, PHP and Ruby.
- The basic concept is that instead of writing a program, you create a class, which is a kind of template containing variables and functions. Objects are self-contained instances of that class, and you can get them to interact in fun and exciting ways.

5.1.1 Brief History about OOP

- The object-oriented paradigm took its shape from the initial concept of a new programming approach, while the interest in design and analysis methods came much later.
  - The first object–oriented language was ***Simula*** (Simulation of real systems) that was developed in 1960 by researchers at the Norwegian Computing Center.
  - In 1970, *Alan Kay* and his research group at Xerox PARK created a personal computer named *Dynabook* and the first pure object-oriented programming language (OOPL) - ***Smalltalk***, for programming the *Dynabook*.
  - In the 1980s, *Grady Booch* published a paper titled Object Oriented Design that mainly presented a design for the programming language, ***Ada***. In the ensuing editions, he extended his ideas to a complete object–oriented design method.
  - In the 1990s, *Coad* incorporated behavioral ideas to object-oriented methods.
- The other significant innovations were Object Modelling Techniques (OMT) by *James Rumbaugh* and Object-Oriented Software Engineering (OOSE) by *Ivar Jacobson*.

5.1.2 OOP Definition

- Grady Booch has defined object–oriented programming as:

“A method of implementation in which programs are organized as ***cooperative collections of objects****, each of which represents an* ***instance of some class****, and whose classes are all members of a* ***hierarchy*** *of classes united via* ***inheritance relationships****”.*

5.1.3 What are the are the differences between Procedural and Object Oriented Programming?

- **Procedural Programming** can be defined as a programming model which is based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program's execution, any given procedure might be called at any point, including by other procedures or itself. Example of programming languages: BASIC, Pascal and C.
- **Object oriented programming** can be defined as a programming model which is based upon the concept of objects. Objects contain data in the form of attributes and code in the form of methods. In object oriented programming, computer programs are designed using the concept of objects that interact with real world. Object oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types. Example of programming and scripting languages: Java, C++, C#, Python, PHP, Ruby, Perl, Objective-C, Swift, Scala.

5.1.3 What are the are the differences between Procedural and Object Oriented Programming?

PROCEDURAL ORIENTED PROGRAMMING	OBJECT ORIENTED PROGRAMMING
In procedural programming, program is divided into small parts called ***functions***.	In object oriented programming, program is divided into small parts called ***objects***.
Procedural programming follows ***top down approach***.	Object oriented programming follows ***bottom up approach***.
There is ***no access specifier*** in procedural programming.	Object oriented programming have access specifiers like ***private***, ***public***, ***protected*** etc.
Adding new data and function is not easy.	Adding new data and function is easy.
Procedural programming have simple way for hiding data so it is ***less secure***.	Object oriented programming provides good ways data hiding so it is ***more secure***.
In procedural programming, *overloading is not possible*.	***Overloading*** is possible in object oriented programming.
In procedural programming, ***function*** is more important than data.	In object oriented programming, ***data*** is more important than function.
Procedural programming is based on unreal world.	Object oriented programming is based on ***real world***.

5.1.4 Features of OOP

- The important features of object–oriented programming are:
  - Bottom–up approach in program design
  - Programs organized around objects, grouped in classes
  - Focus on data with methods to operate upon object's data
  - Interaction between objects through functions
  - Reusability of design through creation of new classes by adding features to existing classes

5.2 OOP Basic Terminologies

- **Class**: A class is a category of objects, classified according to the members that they have. It is *Pattern* or *blueprint* for creating an object. A class contains all attributes and behaviors that describe or make up the object.
- **Object and Instances**: An instance of a particular class; an Object is the logical view and called Instance when be exist in memory.
- **Members**: Objects can have their own data, including variables and constants, and their own methods. The variables, constants, and methods associated with an object are collectively referred to as its *members* or *features*.
- **Attributes or properties**: which represent some of the features of a class datatype, it is may be primitive datatypes or objects from other classes
- **Behaviors**: it is represents the other part of features of the class and represent the operations which work on the attributes of the same class; i.e. it is the functions belongs to the class and called *methods*.

5.2 OOP Basic Terminologies

- **Message**: All the objects from the same class have the same methods and attributes, when you want to call a method you have to decide which object will call the method; i.e. you send a message to the object by calling a method to work on its attributes. The general form of message is: ***object_Name.member_Name()***
- **Encapsulation**: Encapsulation refers to mechanisms that allow each object to have its own data and methods. OOP has mechanisms for restricting interactions between components. So, some members may be ***private*** - it is accessible only by the methods of its class - and some other members may be ***public*** - it is accessible outside and inside the class through any object of that class.
- **Abstraction**: Refers to hiding the internal details of an object from the user, and its class is independent on the application which that object is used in.
- **Polymorphism**: Generally, the ability of different classes of object to respond to the same message in different, class-specific ways. Polymorphic methods are used which have one name but different implementations for in the same class or different classes. There are two types of polymorphism: ***Overloading***, and ***Overriding***.

5.2 OOP Basic Terminologies

- **Overloading**: Allowing the same method name to be used for more than one implementation in the same class, with different inputs. It has two types: ***Operator*** Overloading, and ***Function(Method)*** Overloading.
- **Function(method) Overloading**: Two or more methods with the same name defined within a class are said to be overloaded. This applies to both *constructors* and other *methods*.
- **Operator Overloading**: It is the mechanism to enrich the functionality of an operator defined in a programming language to deals among objects of a class, by define functions as members in that class to overload operator’s functionalities.
- **Inheritance (*is-a*)**: Refers to the capability of creating a *new class* from an existing *class*. It is a relationship between classes where one class is a *parent (super or base)* of another *(Child, subclass, derived)*. It implements *"is-a"* relationships between objects. Inheritance takes advantage of the commonality among objects to reduce complexity.

5.2 OOP Basic Terminologies

- **Inheritance Hierarchy**: The relationship between _super classes_ and _subclasses_ is known as an inheritance hierarchy.
- **Multi-Level Inheritance**: In Multilevel Inheritance a derived class can also inherited by another class.
- **Multiple Inheritance**: The ability of a class to extend more than one class; i.e. A _class_ can inherit characteristics and features from more than one parent class.
- **Overriding**: A _method_ defined in a _superclass_ may be overridden by a _method_ of the same name defined in a _subclass_. The two _methods_ must have the same name and number and types of formal input parameters.
- **Constructor**: A constructor is an _instance_ _method_ that has the following Characteristics:
  - It has the same name as the class
  - It is auto calling when an object is created to be the initializing behavior of its life.
  - It can be overloaded, i.e. any class may have more than one constructor.
  - It can not return any datatype.

5.2 OOP Basic Terminologies

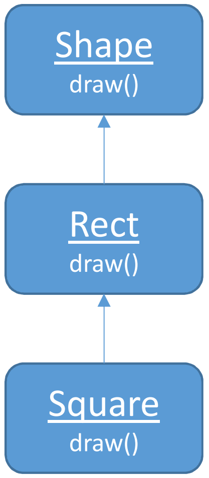
- **Destructor**: A destructor is an *instance* *method* that has the following Characteristics:
  - It has the same name as the class with *tilde* "~" character at the beginning of its name.
  - It is auto calling before during the destruction of an object, i.e. it is the *last behavior* of object's life. Actions executed in the destructor include the following:
    - *Recovering the heap space allocated during the lifetime of an object*
    - *Closing file or database connections*
    - *Releasing network resources*
  - It can not be overloaded, i.e. any class has only one destructor.
  - It can not return any datatype.
- **Static Members (Class members)**: A class can also hold data *variable* and *constants* that are *shared by* all of its objects and can handle *methods* that deal with an *entire class rather than an individual object.* These members are called ***class members*** or, in some languages (C++ and Java, for example), ***static members***. The members that are associated with objects are called ***instance members (Non-static members)***.

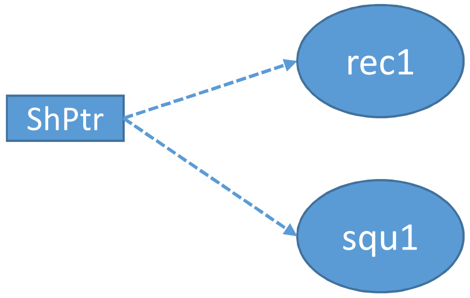
5.2 OOP Basic Terminologies

- **Association (*has-a*)**: Association establish relationship between two separate <u>*classes*</u> through their <u>*objects*</u>. The relationship can be *one to one*, *one to many, many to one and many to many*. Association is a relationship among classes which is used to show that instances of classes could be either ***linked to each other*** or combined *logically* or *physically* into some <u>*aggregation*</u>.
- **Weak Association (*Aggregation*)**: It is a specialized form of association between two or more objects in which the objects have their own life-cycle but there exists an ownership as well. As an example, an employee may belong to multiple departments in the organization. However, if the department is deleted, the employee object wouldn't be destroyed.
- **Strong Association (*Composition*)**: Composition is a specialized form of association in which if the container object is destroyed, the included objects would <u>*cease to exist*</u>. It is actually a ***strong*** type of association and is also referred to as a "***death***" relationship. As an example, a house is composed of one or more rooms. If the house is destroyed, all the rooms that are part of the house are also destroyed as they cannot exist by themselves.

5.2 OOP Basic Terminologies

- **Dynamic (Late) Binding**: Dynamic binding also called dynamic dispatch is the process of linking method call to a specific sequence of code at run-time. It means that the code to be executed for a specific method call is not known until run-time.





5.2 OOP Basic Terminologies

- **Virtual Function**: A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function. It is the mechanism of ***Dynamic Binding*** concept.
- **Abstract Method (Pure Virtual Function)**: An abstract method is a *method* that is declared, but contains no implementation, without body. It is needed to be overridden by the subclasses for standardization of generalization. (Ex: Even Handling).
- **Abstract Class**: Abstract classes are distinguished by the fact that you may not directly construct objects from them. An abstract class may have one or more *abstract methods*. Abstract classes may not be instantiated, and require *subclasses* to provide implementations for the abstract methods. It is needed for standardization or generalization.
- **Concrete Class**: Any class can be initiating objects directly from it, it is not abstract class.

5.3 OOP Fundamentals Characteristics

- Alan Kay, considered by some to be the father of object-oriented programming, identified the following such as fundamental to OOP:

1. Everything is an object
2. Each object has its own memory, which consists of other objects.
3. Every object is an instance of a class, i.e. any object has a particular type.
4. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action bundled with whatever arguments may be necessary, to complete the task.
5. The class is the repository for behavior associated with an object. That is, all object that are instances of the same class can perform the same action, i.e. can receive the same messages.
6. Classes are organized into a singly rooted tree structure, called the ***inheritance hierarchy***.

5.4 Advantages of OOP

- Re-usability: It means reusing some facilities rather than building it again and again. This is done with the use of a class. We can use it ‘n’ number of times as per our need.
- Simplicity: Object oriented programming is based on ***real world***.
- Modularity: Parts of a program can be stopped without made a problem with other parts.
- Extensibility: Adding new functionalities to the program is easy because of modularity.
- Modifiability: Any module can be modified easy.
- Maintainability: Any problem can be cached easy and solved because of modularity.

5.5 Complex Class Example – Using struct

- We will write a class represent the Complex number:

```
struct Complex
{     private:
           float real;
           float imag;
      public:
           Complex();       // constructor without parameters 'default constructor'
           Complex(float r);     // constructor with one input parameter
           Complex(float r, float i);// constructor with two input parameters
           ~Complex();      // Destructor; only one for each class
           void setReal(float r) ;    // Setter for the real attribute
           void setImag(float i) ;    // Setter for the imag attribute
           float getReal() ;          // getter for the real attribute
           float getImag() ;          // getter for the imag attribute
           Complex add(Complex c);    // a member to perform addition behavior
           Complex sub(Complex c);    // a member to perform subtraction behavior
           void print();              // a member to perform printing behavior
};
```

5.5 Complex Class Example – Using struct

```
        Complex::Complex()
{
        real = image = 0 ;
        cout<<"\n Default Constructor is calling";
}
        Complex::Complex(float r, float i)
{
        real =r;        imag = i ;
        cout<<"\n Constructor with two parameters is calling";
}
        Complex::Complex(float r)
{
        real = imag = r;
        cout<<"\n Constructor with one parameters is calling";
}
        Complex::~Complex()
{
        cout<<"\n Destructor is calling";
}
```

5.5 Complex Class Example – Using struct

```
void Complex::setReal(float r)
{
     real = r ;
}
void Complex::setImag(float i)
{
     imag = i ;
}
float Complex::getReal()
{
     return real ;
}
float Complex::getImag()
{
     return imag ;
}
```

5.5 Complex Class Example – Using struct

```
Complex Complex::add(Complex c)
{
    Complex temp;
    temp.real = real + C.real;
    temp.imag = imag + c.imag;
    return temp;
}
Complex Complex::sub(Complex c)
{
    Complex temp;
    temp.real = real - C.real;
    temp.imag = imag - c.imag;
    return temp;
}
```

5.5 Complex Class Example – Using struct

```
void Complex::print()
{
    if(imag<0)
    {
        cout<<real<<" - "<<fabs(imag)<<"i"<<endl;
    }
    else
    {
        cout<<real<<" + "<<imag<<"i"<<endl;
    }
}
```

5.5 Complex Class Example – Using struct

```
int main()
{
    clrscr() ;
     Complex myComp1, myComp2(3, 4), resultComp(5) ;
     myComp1.setReal(7) ;
     myComp1.setImag(2) ;
     resultComp = myComp1.add(myComp2) ;
     resultComp.print() ;
     resultComp = myComp1.sub(myComp2) ;
     resultComp.print() ;
     return 0 ;
}
```

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Assignments

- Complex Example with "add" and "subtract" member functions.
- Note: the following points will cover in the lab (***ask your lab assistant***):
  - Moving from .C to .CPP compiler:
    - Flexible Deceleration of variables anywhere in the program.
    - Eliminating "void" keyword from brackets in function headers.
    - cin>> and cout<< streams.
    - Dynamic allocation with "new" and "delete" operators.
    - Call By Reference, with the Swap example.
  - Emphasize on Member Functions VS. Stand Alone Functions, and scope resolution operator.
  - Explain the Inline Function.

Polymorphism
Function Overloading

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

Content

- Class Vs Struct
- Function overloading: setAll function with 3 implementation in class complex
- "***this***" pointer
- Stack Example
- Static variables and static methods
- Friend function
- Passing an object as function parameter: Stack Example on passing objects, "viewContent( )" friend function.

6.1 Using Class instead of Struct

- Actually the structure in C++ with added features like private and public sections, and including functions, was made for helping C programmers to move from structure programming to object-oriented programming.
- The only difference between a struct and class in C++ is the default accessibility of member variables and methods. In a struct they are public; in a class they are private.

6.2 Polymorphism: Function Overloading

- As we discussed in the example of Complex example, there was three constructors, default constructor (without parameters), constructor with one parameter, and the third one with two parameters. It is one of the function overloading example.
- This may be done for any of the other behaviors.
- We have a behavior for setting the real and imag attributes when we call it, called setALL(). We may make three methods with the same name but with different parameters, as follow

```
void setAll();
void setAll(float f);
void setAll(float r, float i);
```

- That means the same behavior is performed but with different methodologies.
- Rewrite the Complex example with using class instead od struct and add the setAll().

6.2 Polymorphism: Function Overloading

```
class Complex
{          // The default accessibility in class is private so if we do not
           // label the section with accessibility type, it will be private.
           float real;
           float imag;
      public:
           Complex();     // constructor without parameters default constructor
           Complex(float r);              // constructor with one input parameter
           Complex(float r, float i);     // constructor with two input parameters
           ~Complex();                // Destructor; only one for each class
           void setReal(float r) ;    // Setter for the real attribute
           void setImag(float i) ;    // Setter for the imag attribute
```

6.2 Polymorphism: Function Overloading

```
class Complex
{          …
           …
     float getReal() ;         // getter for the real attribute
     float getImag() ;         // getter for the imag attribute
     void setAll();            // setting both of real and imag with 0
     void setAll(float f);     // setting both of real and imag with f
     void setAll(float r, float i); // setting real and imag with r & i
     Complex add(Complex c);   // a member to perform addition behavior
     Complex sub(Complex c);   // a member to perform subtraction behavior
     void print();             // a member to perform printing behavior
};
```

6.2 Polymorphism: Function Overloading

```
void Complex::setAll()
{
     real = imag = 0;
}
void Complex::setAll(float f)
{
     real = f;
     imag = f;
}
void Complex::setAll(float r, float i)
{
     real = r;
     imag = i;
}
```

6.3 The “*this*” pointer

- Every object in C++ has access to its own address through an important pointer called ***this*** pointer.
- To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class:
  1. Each object gets its own copy of the data member.
  2. All objects access the same function definition as present in the code segment.
- Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
- So, the “***this”*** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object by address.
- The stand alone and friend (will be explained later) functions do not have a ***this*** pointer, because they are not members of a class. Only member functions have a ***this*** pointer.

6.3 The “*this*” pointer

```
        Complex::Complex(float real, float imag)
{
        this->real = real;
        this->imag = imag;
}
void    Complex::setAll(float f)
{
    this->real = f;
    this->imag = f;
}
```

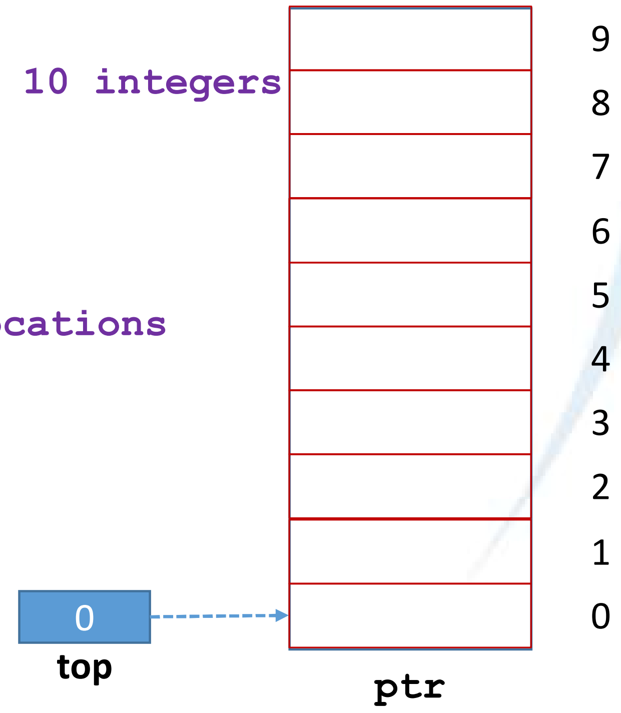
6.4 Stack Example

- We will make a class to represent a stack data structure (array based) with dynamic allocation to the array places.

```
class Stack
{
  private:
     int top ;     // indicator to the top of stack
     int size;     // the max size of the stack
     int *ptr;     // the pointer to create and access the stack elements
```

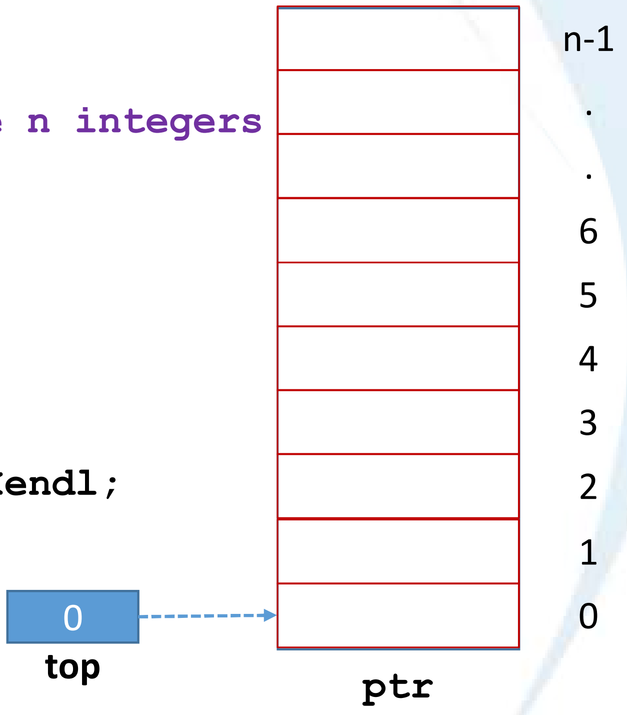
6.4 Stack Example

```
public:
   int isFull() ;  //functions using to check, they offer no service
   int isEmpty(); //to the outside world.
   Stack()  // The default constructor with stack size is 10 integers
   {
        top = 0 ;       // initialize stack state
        size = 10;
        ptr = new int[size];     // Allocate the stack locations

        cout<<"This is the default constructor"<<endl;
   }
```



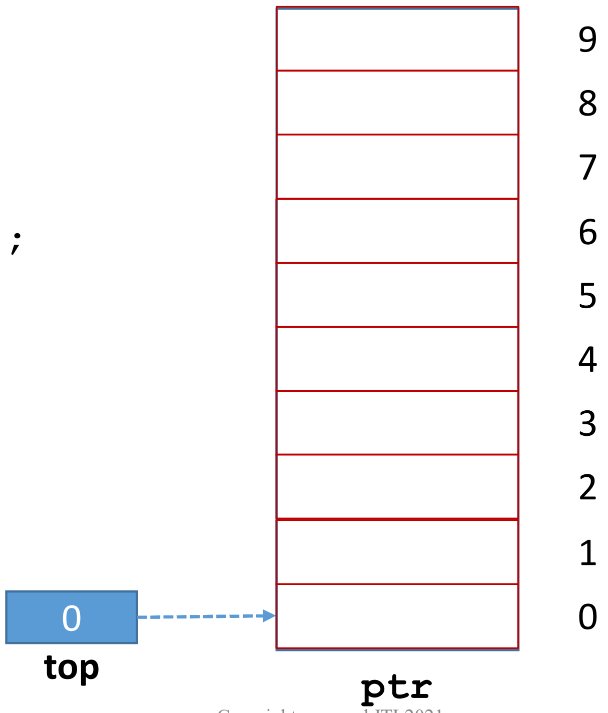
6.4 Stack Example

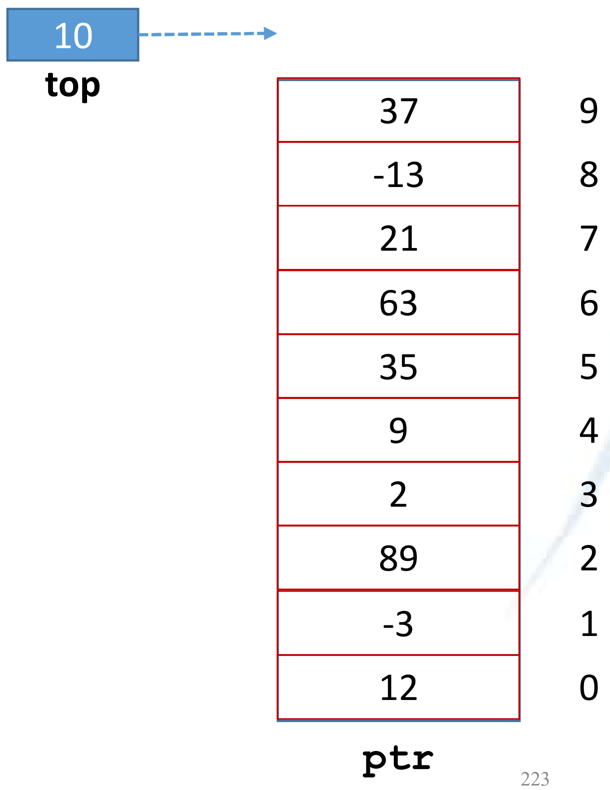
```
Stack(int n) // The default constructor with stack size n integers
{
     top = 0;
     size = n;
     ptr = new int[size];

     cout<<"This is a constructor with one parameter"<<endl;
}
~Stack()
{
     delete[] ptr;  size=0;

     cout<<"This is the destructor"<<endl;
}
```



6.4 Stack Example

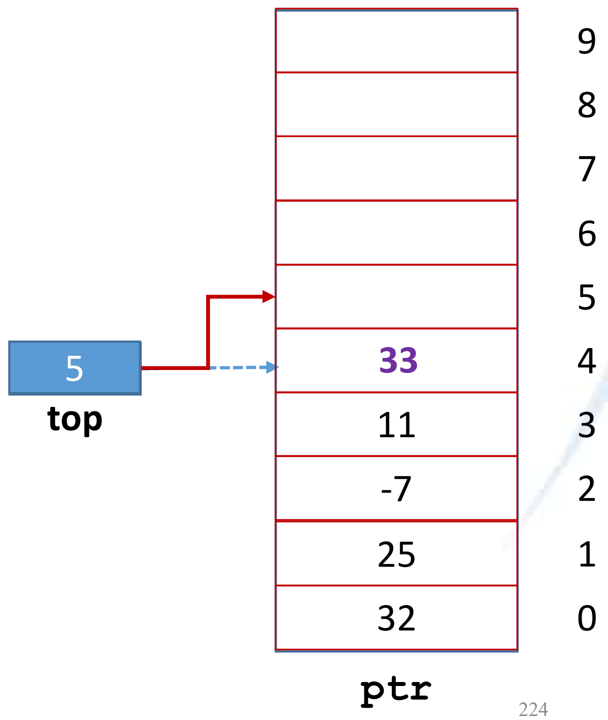
```
     int push(int n);
     int pop(int & n);
};

int Stack::isFull()
{
     return (top==size) ;
}

int Stack::isEmpty()
{
     return (top==0) ;
}
```





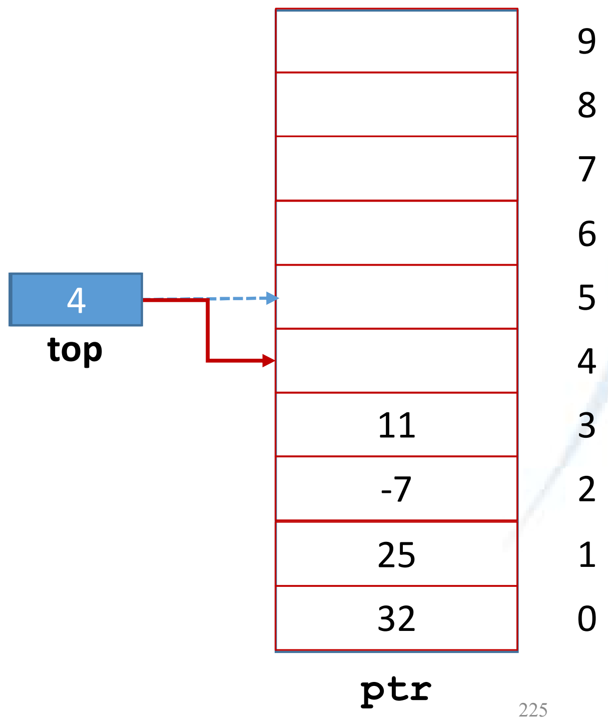
6.4 Stack Example

```
int Stack::push(int n)
{
     if (isFull())
          return 0;
     ptr[top] = n;
     top++;
     return 1
}
```



6.4 Stack Example

```
int Stack::pop(int& n)
{
     if (isEmpty())
          return 0;
     top--;
     n = ptr[top];
     return 1;
}
```



6.5 Static Variables and Static Methods

- We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
- A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by re-declaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.
- Let's apply that on the Stack example, we will define a static private **int** variable identified as **counter**, to count the exist objects currently in memory, for accessing it we have to make a method called **getCounter** in public section, and it will be static for accessing at any time *before/after* creating any object.
- The changing of the counter static variable will be done by the constructors (increment it) and destructor (decrements it).

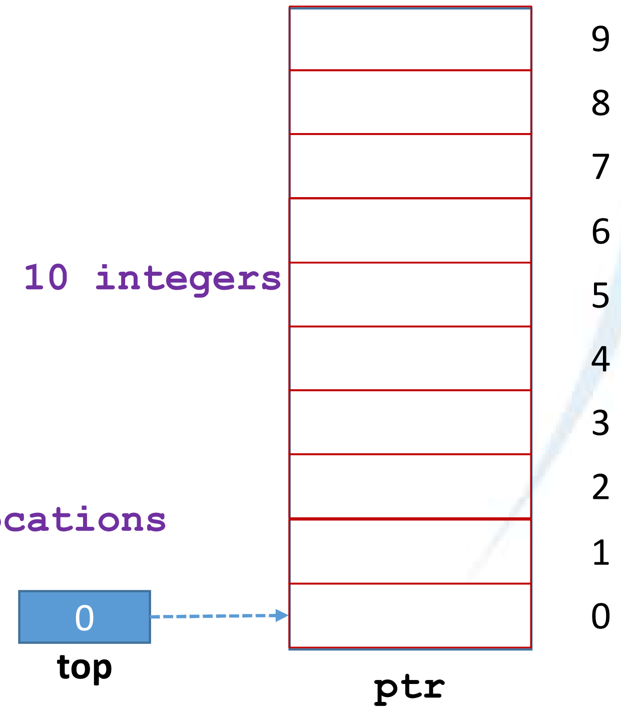
6.5 Static Variables and Static Methods

- We will make a class to represent a stack data structure (array based) with dynamic allocation to the array places.

```
class Stack
{
  private:
     int top ;       // indicator to the top of stack
     int size;       // the max size of the stack
     int *ptr;       // the pointer to create and access the stack elements
     static int counter; // it is not allowed to initialize it here in C++
```

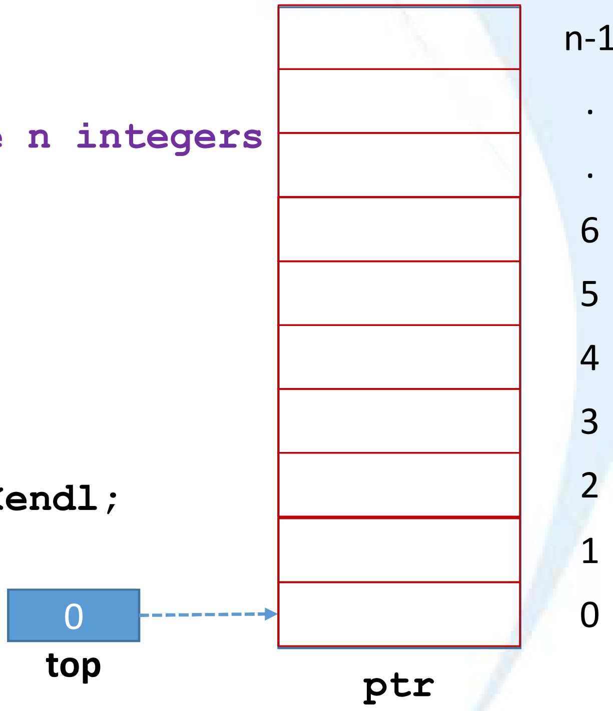
6.5 Static Variables and Static Methods

```
public:
   int isFull() ;  //functions using to check, they offer no service
   int isEmpty(); //to the outside world.
   static int getCounter()
   {
        return counter;
   }
   Stack()  // The default constructor with stack size is 10 integers
   {
        top = 0 ;      // initialize stack state
        size = 10;
        ptr = new int[size];     // Allocate the stack locations
        counter ++;
        cout<<"This is the default constructor"<<endl;
}
```



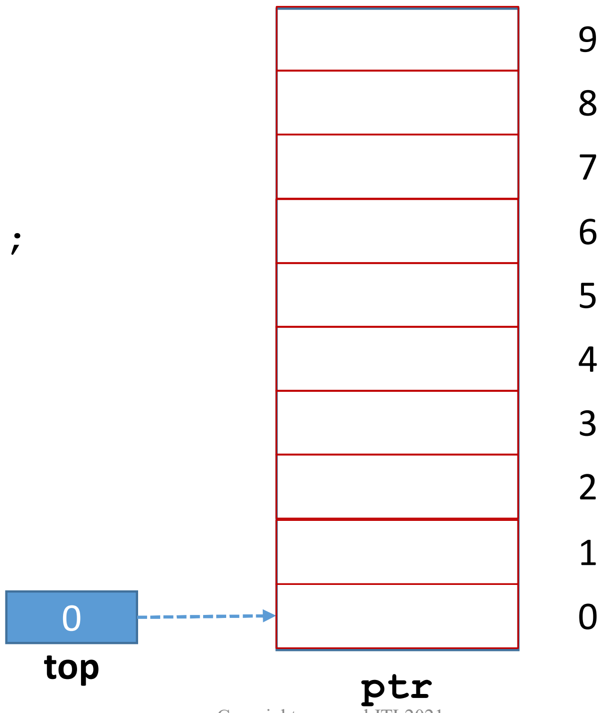
6.5 Static Variables and Static Methods

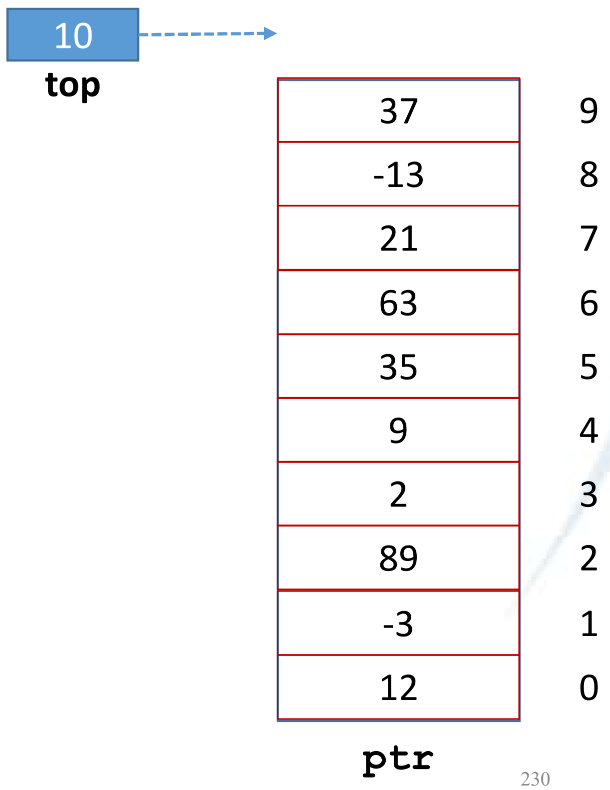
```
Stack(int n) // The default constructor with stack size n integers
{
     top = 0;
     size = n;
     ptr = new int[size];
     counter ++;
     cout<<"This is a constructor with one parameter"<<endl;
}
~Stack()
{
     delete[] ptr;  size=0;
     counter --;
     cout<<"This is the destructor"<<endl;
}
```



6.5 Static Variables and Static Methods

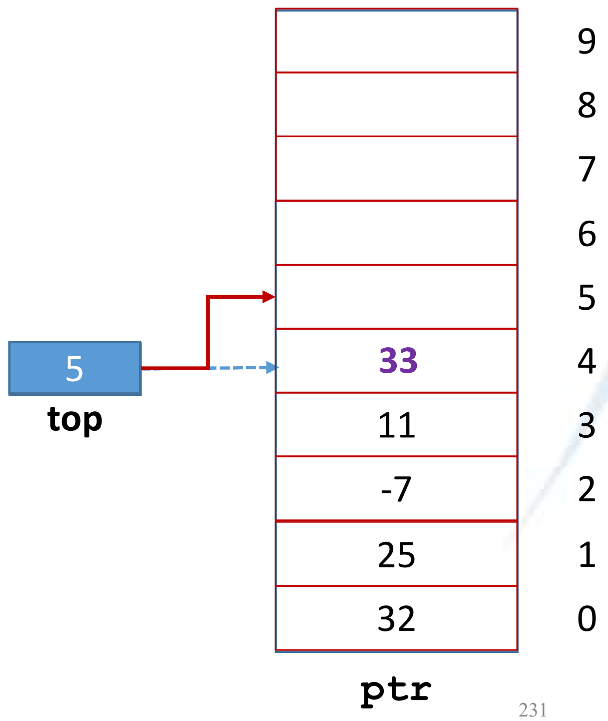
```
     int push(int n);
     int pop(int & n);
};

int Stack::isFull()
{
     return (top==size) ;
}

int Stack::isEmpty()
{
     return (top==0) ;
}
```





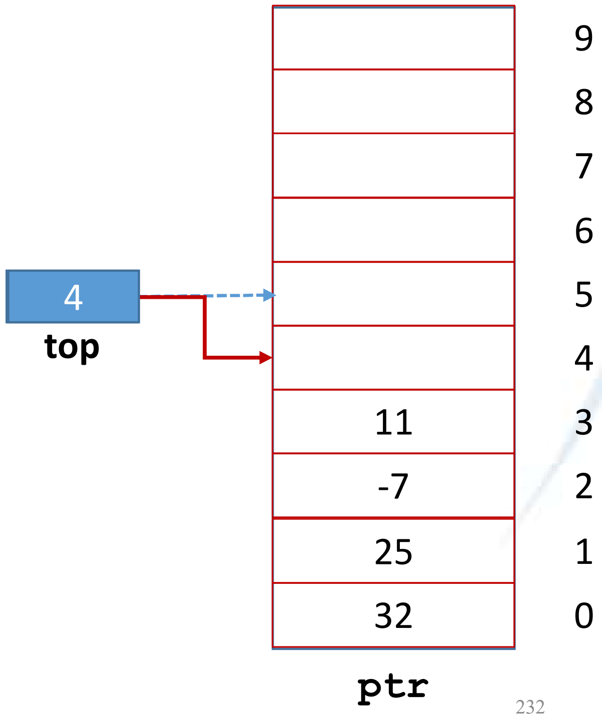
6.5 Static Variables and Static Methods

```
int Stack::push(int n)
{
     if (isFull())
          return 0;
     ptr[top] = n;
     top++;
     return 1
}
//static variable initialization
int Stack::counter = 0;
```



6.5 Static Variables and Static Methods

```
int Stack::pop(int & n)
{
     if (isEmpty())
          return 0;
     top--;
     n = ptr[top];
     return 1;
}
```



6.5 Static Variables and Static Methods

```
int main()
{
     clrscr() ;
     int num ;
     Stack s1(2) ;
     s1.push(5);
     s1.push(14);
     s1.push(20) ;      // Stack is full so the third one will not in
     if(s1.pop(num))
     {
          cout<<num<<endl ; // the result will be 14
     }
```

6.5 Static Variables and Static Methods

```
    if(s1.pop(num))
    {
        cout<<num<<endl ; // It will print 5 last element
    }
    if(s1.pop(num))
    {
        cout<<num<<endl ; // The stack is empty
    }
    else cout<< "\n Stack is empty …";
    getch() ;
    return 0;
}
```

6.5 Static Variables and Static Methods

```
int main()
{
     cout<<"\nThe number of objects created = "<<Stack::getCounter();
     Stack s1, s2(5);
     cout<<"\nThe number of objects created = "<<Stack::getCounter();
     {
          Stack s3(10);
        cout<<"\nThe number of objects created = "<<Stack::getCounter();
     }
     cout<<"\n the number of objects created = "<<Stack::getCounter();
     getch();
     return 0;
}
```

6.6 Friend Function

- A friend function of a class is defined outside that class' scope but it has the right to access all private members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- A friend can be a stand alone function, or member function, in which case the entire class and all of its members are friends.
- The friend function ***violates*** an important concept in Object-Oriented programming; “Encapsulation”.
- To declare all member functions of class *ClassTwo* as friends of class *ClassOne*, place the following declaration in the definition of class *ClassOne*

```
    friend class ClassTwo;
```

- Let’s make a friend function to class Stack, for viewing the contents of the stack without pop its data.

6.6 Friend Function

```
Class Stack{
     private:
     int top ;       // indicator to the top of stack
     int size;       // the max size of the stack
     int *ptr;       // the pointer to create and access the stack elements
     static int counter;  // it is not allowed to initialize it here in C++
public:
     int isFull() ;  //functions using to check, they offer no service
     int isEmpty();  //to the outside world.
     static int getCounter();
     Stack();
     Stack(int n);
     ~Stack();
     int push(int n);
     int pop(int &n);
     friend void viewContent(Stack s);
};
```

6.6 Friend Function

```
void viewContent (Stack s)
{
     for(int i = 0; i<s.top ; i++)
          cout<<"\n Element no ("<<i+1<<") ="<< s.ptr[i];
}
```

- As we show, there is a non-member function can access the private members of a class, just it is declared as a friend function inside that class.

6.6 Friend Function

```
int main()
{    clrscr() ;
     int num ;
     Stack s1(5) ;
     s1.push(5);
     s1.push(14);
     s1.push(20) ;
     viewContent(s1); // All the contents are displayed
     if(s1.pop(num))
     {
          cout<<num<<endl ; // the result will be 20
     }
     getch();
     return 0;
}
```

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Assignments

- Continue Complex (constructors and setters overloading).
- Stack Example:
  - Constructors.
  - Destructor.
  - Accessing members
  - Static counter and static getCounter
  - main( ) test.
  - Write the "viewContent( )" friend function.

- Note: You should Trace the code.

Dynamic Area Problem, Copy Constructor, and Operator Overloading

Java™ Education and Technology Services

Invest In Yourself , Develop Your Career

Content

- Dynamic Area Problem and its solution
  - Using Call by Reference.
  - Using the Copy Constructor.
- Copy constructor
  - Example on Stack
- Operator Overloading
  - Example on Class Complex: +, ++, =, +=, ==, casting

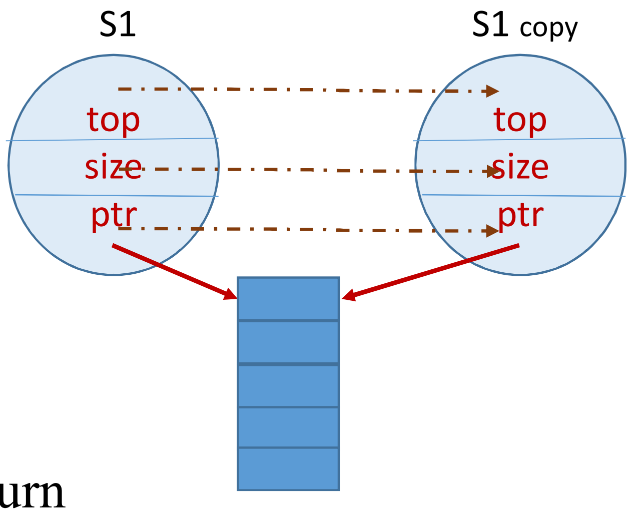
7.1 Dynamic Area Problem

- When we pass an object to a function as call by value, a copy of the object is passed to the function. How this copy is created?
- Actually, the C++ compiler make another constructor by default– if we do not write it- called *Copy Constructor*, it is calling when a copy of an object is needed – like pass the object by value or return it by value -, it makes a shallow copy, i.e. byte wise copy.
- If the object creation make dynamic memory allocation, it causes a problem when pass the object by value or return it by value, Why? …
- When the function return, all the local variables and arguments are removed from memory, so the shallow copy of the passed object will removed then the destructor is calling for that shallow copy, here the shallow destructing will deallocate the memory locations which is shared with the original object.

7.1 Dynamic Area Problem

- Lets see what happen when the object of a Stack class are passed to the viewContent function:

  viewContent(S1)

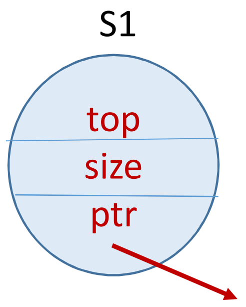


- When the function return

7.1 Dynamic Area Problem

- Lets see what happen when the object of a Stack class are passed to the viewContent function:

  viewContent(S1)



- When the function return

7.1 The solution of Dynamic Area Problem

- There are two ways to solve that problem:
  - Make the passing to an object by reference instead of call by value, or
  - Write a copy constructor to make a concrete (deep) copy of the object to shallow copy.
- The side effect of send an object by reference, the object attributes may be changed by the function – specially if the function id friendly function as Stack example-.
- The side effect of copy constructor solution that the numbers of objects created for calling by value and return by value is overhead on program execution time, i.e. affect on the performance.

7.2 Copy Constructor

- A copy constructor is a member function which initializes an object using another object of the same class.
- The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
- If a copy constructor is not defined in a class, the compiler itself defines one, which make a shallow copy of object by make the byte wise equality.
- If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.
- The general form of the copy constructor as follow:

```
        ClassName (ClassName & obj){}
```

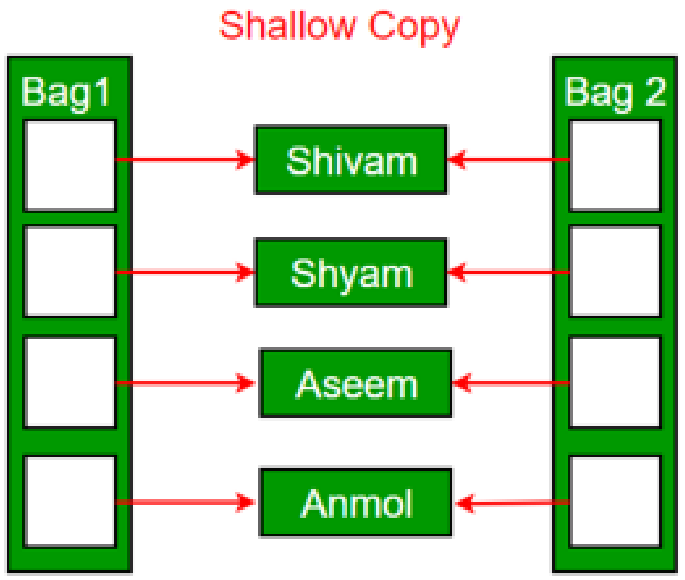
- That means, the copy constructor has only one parameter which is a reference to an object from the same class.

7.2 Copy Constructor

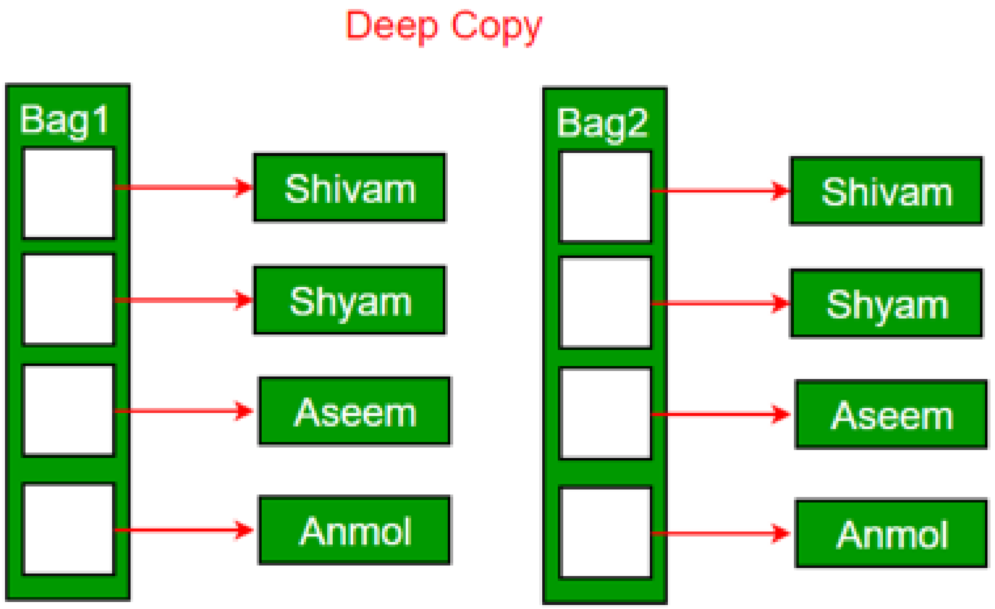
- The Copy Constructor is auto calling in the following cases:
  - When an object of the class is returned by value.
  - When an object of the class is passed (to a function) by value as an argument.
  - When an object is constructed based on another object of the same class.
- Some times the compiler needs to generate a temporary object, so it may call the copy constructor.
- Default constructor does only shallow copy of the object, so at case – like stack class- we need to implement the copy constructor in our class to describe how make a deep copy of an object form that class.

7.2 Copy Constructor

Default constructor does only shallow copy.



Deep copy is possible only with user defined copy constructor.

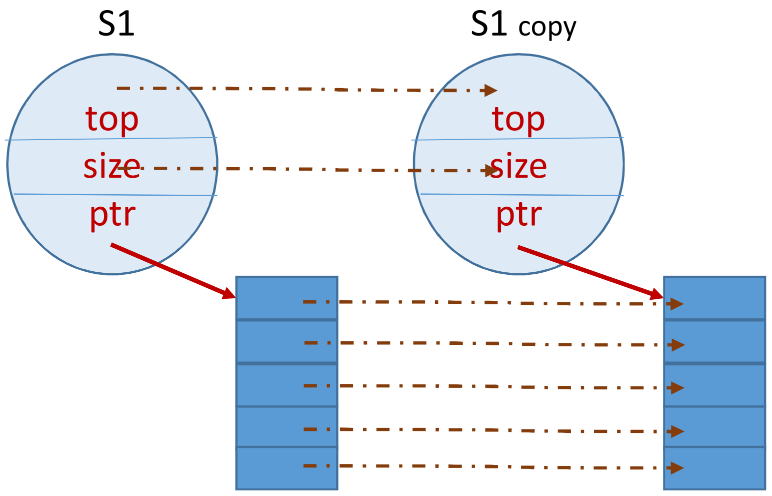


7.2 Copy Constructor: Stack Example

```
Class Stack{
      private:
      int top ;         // indicator to the top of stack
      int size;         // the max size of the stack
      int *ptr;         // the pointer to create and access the stack elements
      static int counter;     // it is not allowed to initialize it here in C++
public:
      int isFull() ;  //functions using to check, they offer no service
      int isEmpty();    //to the outside world.
      static int getCounter();
      Stack();
      Stack(Stack & s);
      Stack(int n);
      ~Stack();
      int push(int n);
      int pop(int &n);
      friend void viewContent(Stack s);
};
```

7.2 Copy Constructor: Stack Example

```
Stack(Stack & s)
{
    top = s.top
    size = s.size;
    ptr = new int[size];
    for(int i=0;i<top;i++)
        ptr[i] = s.ptr[i];
}
```



7.2 Copy Constructor

- **How to create an object which is initialized by another object of the same type:**

```
int main()
{    int num
     Stack s1(10);
     s1.push(7);
     s1.push(13);
     s1.push(-5);
     Stack s2(s1);
     s2.pop(num);
     cout<<num<<endl;
     s1.pop(num);
     cout<<num<<endl;
     getch();
     return 0;
}
```

7.3 Operator Overloading

- In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**.
- A feature in C++ that enables the redefinition of operators. This feature operates on user defined objects. All overloaded operators provides syntactic sugar for function calls that are equivalent. Without adding to / changing the fundamental language changes, operator overloading provides a pleasant façade.
- It is a type of polymorphism in which an operator is overloaded to give user-defined meaning to it. Overloaded operators is used to perform operation in user-defined datatype.
- Operator Overloading means providing multiple definition for the same operator.
- Operator overloading is a specific case of polymorphism in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments. It can easily be emulated using function calls.

7.3.1 Rules for operator overloading in C++

- In C++, following are the general rules for operator overloading.
- Only built-in operators can be overloaded. New operators can not be created.
- Arity (arguments or operands) of the operators cannot be changed.
- Precedence and associativity of the operators cannot be changed.
- Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.
- Operators cannot be overloaded for built in types only. At least one operand must be used defined type.
- 6) Assignment (=), subscript ([]), and function call (“()”) operators must be defined as member functions
- 7) Except the operators specified in point 6, all other operators can be either member functions or a non member (stand alone) functions.

7.3.2 Operators that cannot be overloaded in C++

- In C++ we can overload some operators like +, -, [], = etc. But we cannot overload any operators in it. Some of the operators cannot be overloaded. These operators are like below:
  - "." Member access or dot operator
  - "? : " Ternary or conditional operator
  - "::" Scope resolution operator
  - ".*" Pointer to member operator
  - "sizeof" The object size operator
- Note: The short-circuit operators && and || is not affected by the overloading.
- Now we will make operator overloading on Complex class

7.4.1 Mathematical Operator: +

```
class Complex
{       ...
        public:
                Complex(Complex & c);   // Copy constructor
                Complex operator + (Complex & c) // using in form: Com1 + com2
                {
                        Complex temp(real+c.real, imag+c.imag);
                        return temp;
                        // return Complex(real+c.real, imag+c.imag);
                }
                Complex operator + (float f) // using in form: Com1 + f
                {
                        Complex temp(real+f, imag);
                        return temp;
                        // return Complex(real+f, imag);
                }

};
```

7.4.1 Mathematical Operator: +

- What we do if we want to add a complex object to a float number?
- In the above cases there is an object which call the operator overloaded method, but in this case the float variable needs to call an overloaded function; which has to be stand alone to inform the compiler how to use the operator + between float and complex.
- If we need this stand alone function to deal directly with the private members of the complex class, make it friend function.

7.4.1 Mathematical Operator: +

```
class Complex
{       ...
        public:
                Complex(Complex & c);    // Copy constructor
                Complex operator + (Complex & c) // using in form: Com1 + com2
                {
                        Complex temp(real+c.real, imag+c.imag);
                        return temp;
                        // return Complex(real+c.real, imag+c.imag);
                }
                Complex operator + (float f) // using in form: Com1 + f
                {
                        Complex temp(real+f, imag);
                        return temp;
                        // return Complex(real+f, imag);
                }
                friend Complex operator +(float f, Complex & C);

};
```

7.4.1 Mathematical Operator: +

```
Complex operator+ (float f, Complex & c)
{
    Complex temp(f+c.real, c.imag);
    return temp;
    // return c+f;
}
```

7.4.1 Mathematical Operator: +

```
int main()
{
    Complex c1(12, 7),c2(10, -5);
    Complex c3;
    c3=c1+c2;
    c3.print();
    c3=c1+13.65;
    c3.print();
    c3=6.2+c2;
    c3.print();
    getch();
    return 0;
}
```

7.4.2 Unary Operator: ++

```
class Complex
{     ...
     public:
          Complex operator++ ()// prefix
          {     real++;
                imag++;
                return *this;
          }
          Complex operator++ (int dumy)// postfix
          {     Complex temp(*this);
                real++;
                imag++;
                return temp;
          }
};
```

7.4.2 Unary Operator: ++

```
int main()
{
    Complex c1(12, 7),c2(10, -5),c3;
    c3=++c1;
    c1.print();
    c3.print();
    c3=c2++;
    c2.print()
    c3.print();
    getch();
    return 0;
}
```

7.4.3 Assignment Operators: =

```
class Complex
{   ...
    public:
        Complex& operator= (Complex & c)// for cascading =
        {
            real=c.real;
            imag=c.image;
            return *this;
        }

};
```

7.4.3 Assignment Operators: =

```
int main()
{
    Complex c1(9, 7), c2(13), c3;
    c3=c1;
    c1.print();
    c3.print();
    c2=c1=c3;
    c1.print();
    c2.print();
    c3.print();
    getch();
    return 0;
}
```

7.4.3 Assignment Operators: +=

```
class Complex
{   ...
    public:
        Complex& operator+= (Complex & c)
        {
            real+=c.real;
            imag+=c.image;
            return *this;
        }

};
```

7.4.3 Assignment Operators: =

```
int main()
{
    Complex c1(9, 7), c2(13), c3;
    c2+=c1;
    c1.print();
    c2.print();
    c2=c1+=c3;
    c1.print();
    c2.print();
    c3.print();
    getch();
    return 0;
}
```

7.4.4 Comparison Operators: ==

```
class Complex
{    ...
     public:
          int operator== (Complex & c)
          {
               if((real==c.real) && (imag==c.image))
                         reurn 1;
               return 0;
          // return ((real==c.real) && (imag==c.image));
          }

};
```

7.4.4 Comparison Operators: ==

```
int main()
{
    Complex c1(9, 7), c2(13), c3(9, 7);
    cout<<"c1 equals to c3 ? "<<(c1==c3)<<endl;
    if(!(c1==c2))
        cout<<"c1 is not equal to c2"<<endl;
    getch();
    return 0;
}
```

7.4.5 Unary Operator: casting to float

```
class Complex
{   ...
    public:
        operator float ()
        {
            return real;
        // return sqrt(real*real+imag*imag);
        }

};
```

7.4.5 Unary Operator: casting to float

```
int main()
{
    Complex c1(9, 7), c2(13), c3(9, 7);
    cout<<"The float of c1 "<<float(c1)<<endl;
    float f = (float)c2;
    cout<<"The casting of c2 is: "<<f<<endl;
    getch();
    return 0;
}
```

7.4.6 Shift operators with istream and ostream: cin, cout

```
class Complex
{   ...
    public:
      friend istream& operator>>(istream & in, Complex & c);
      friend ostream& operator<<(ostream & out, Complex & c);

};
```

7.4.6 Shift operators with istream and ostream: cin, cout

```
istream& operator>>(istream & in, Complex & c)
{
    cout<<"\n Enter real part:";
    in>>c.real;
    cout<<"Enter imag part";
    in>>c.imag;
    return in;
}
```

7.4.6 Shift operators with istream and ostream: cin, cout

```
ostream& operator<<(ostream & out, Complex & c)
{
    if(imag<0)
            out<<c.real<<" - "<<fabs(c.imag)<<"i"<<endl;
    else
            out<<c.real<<" + "<<c.imag<<"i"<<endl;
    return out;
}
```

7.4.6 Shift operators with istream and ostream: cin, cout

```
int main()
{
    Complex c1(9, 7), c2(13), c3(9, 7);
    cin>>c2;
    cout<<c2;
    getch();
    return 0;
}
```

7.4.7 Array operator [ ]

```
class MyArray
{
    private:
        int size;
        int * data;
    public:
        MyArray(int size)
        {
            this->size = size;
            data= new int[size];
        }
```

7.4.7 Array operator [ ]

```
    ~MyArray()
    {
        delete [] data;
    }
    int & operator [] (int index)
    {
        if(index<size)
                return data[index];
        else
        {
            cout<<"Array Out of Boundaries Exception";
            exit(1);
        }
    }
};
```

7.4.7 Array operator [ ]

```
int main()
{
    MyArray arr(13);
    for(int i=0; i<12; i++)
        cin>>arr[i];
    for (i=0; i<12; i++)
        cout<<arr[i];
    getch();
    return 0;
}
```

7.4.8 Function operator

```
class Point
{   private:
        int x,y;
    public:
        Point();
        Point(int a,int b);
        void operator()(int a, int b)
        {   x=a;
            y=b;
        }
    void print(){cout<<"\n Point Data: x="<<x<<" y="<<y<<endl;}
};
```

7.4.8 Function operator

```
int main ()
{
    Point p1(12, 20);
    p1.print();
    p1(-5, 132);
    p1.print();
    getch();
    return 0;
}
```

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Assignments

- Continue Stack
  - Try without copy constructor. (to see the extra destructor call).
  - Write the copy constructor, and then try again.
- Continue Complex Example: Overloading of all the other operators, and create a main function to test all the operations.
- Continue Stack Example: Overloading the operator =.

- Note: You should Trace the code.

Association among Classes and Collections of Objects

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Content

- Association Class Relations (has-a relationship)
  - Strong: Example of Point, Rect, Circle.
  - Embedded Objects
  - Constructors chaining
- Collections and Temporary Objects:
  1) Static allocation: single objects and array of objects. (how to specify desired constructors for each object in the array).
  2) dynamic allocation: pointers to single object and array of objects.
- Weak Association:
  - Example of: class Picture associated with Rect, Circle.

8.1 Association Relationship (has-a relation)

- Association establish relationship between two separate <u>*classes*</u> through their <u>*objects*</u>. The relationship can be *one to one*, *one to many, many to one and many to many*. Association is a relationship among classes which is used to show that instances of classes could be either ***linked to each other*** or combined *logically* or *physically* into some <u>*aggregation*</u>.
- In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.
- It represent ***has-a*** relationship among classes.
- For example, if there are two classes, Institute and Employee, the Institute class has a collections of Employee objects.

8.1.2 Strong Association (Composition)

- Composition is a specialized form of association in which if the container object is destroyed, the included objects would *cease to exist*. It is actually a ***strong*** type of association and is also referred to as a "***death***" relationship. As an example, a house is composed of one or more rooms. If the house is destroyed, all the rooms that are part of the house are also destroyed as they cannot exist by themselves.
- It represents *part-of* relationship.
- In composition, both the entities are dependent on each other. When there is a composition between two entities, the composed object cannot exist without the other entity.
- Embedded object is a form of strong association

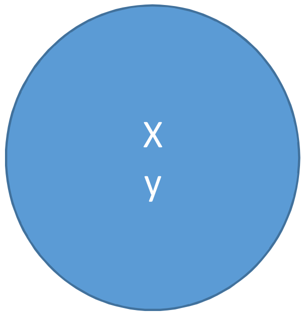
8.1.3 Embedded Objects as Strong Assocaiation

- An embedded object is physically stored in the container object, In other words, the embedded object is actually a part of the container object in which it resides.
- Let's take an example of classes Point, Line, Rect, and Circle
- Any line can be described by two points; start and end.
- And so, any rectangle can be described by two points; upper left and lower right
- Circle is described by central point and radius.

8.1.3 Embedded Objects as Strong Assocaiation

```
class Point
{  private:
       int x ;
       int y ;
  public:
       Point()
       {      x = y = 0 ;
              cout<<"Point default constructor is calling"<<endl;
       }
       Point(int m)
       {      x = y = m ;
              cout<<"Point one parameter constructor is calling"<<endl;
       }
       Point(int m, int n)
       {      x = m ;
              y = n ;
              cout<<"Point two parameter constructor is calling"<<endl;
       }
       ~Point(){cout<<"Point destructor is calling"<<endl;}
```

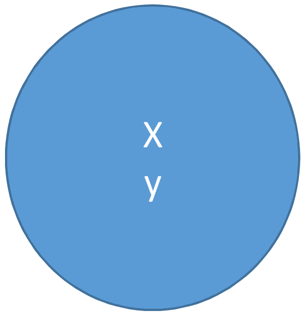
Class Point



8.1.3 Embedded Objects as Strong Association

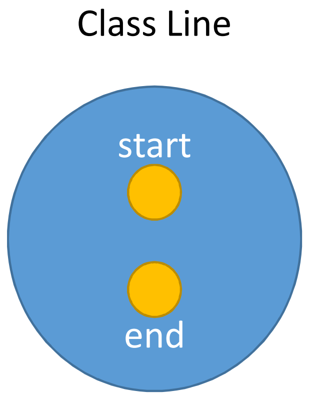
```
        void setX(int m)
        {       x = m ;
        }
        void setY(int n)
        {       y = n ;
        }
        void setXY(int m, int n)
        {       x=m;
                y=n;
        }
        int getX()
        {       return x ;
        }
        int getY()
        {       return y ;
        }
        void print(){cout<<"\n Point Data: x="<<x<<" y="<<y<<endl;}
};
```

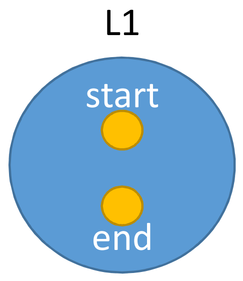
Class Point



8.1.3 Embedded Objects as Strong Association

```
class Line
{  private:
       Point start;
       Point end;
  public:
       Line()
       {      start.setXY(0,0);   end.setXY(0,0);
              cout<<"Line default constructor is calling"<<endl;
       }
       Line(int x1, int y1, int x2, int y2)
       {       start.setXY(x1,y1);      end.setXY(x2,y2);
              cout<<"Line with 4 parameter constructor is calling"<<endl;
       }
       ~Line(){cout<<"Line destructor is calling"<<endl;}
       void print()
       {      cout<<"\nStart:";   start.print();
              cout<<"\nEnd:";     end.print();
       }
};
```





8.1.3.1 Constructor and destructor Chaining in case of embedded objects

- In fact, to create an object from class Line, all the instance attributes must be created and allocated in the memory before any constructor of the Line constructors could be called. So, the two objects from class Point, start and end, are completely created and allocated as apart of Line object, then the constructors for the two Point objects are executed before the constructor of Line is executing.
- And vise versa, when trying to remove a Line object from the memory, the destructor of Line is the last behavior of the lifetime of the Line object, after it is executed, the Line object is starting to remove from the memory; i.e. its components will be removing, so the two Point objects will be removing, then the destructor for these objects is calling for each one after destructor of Line destructor.

8.1.3.1 Constructor and destructor Chaining in case of embedded objects

- The constructor chaining of the an object which has an embedded object, the constructor of the embedded object is execute first and then the constructor of the container object.
- The destructor chaining of the an object has embedded another object, the destructor of the embedded object is execute after the destructor of the container object.

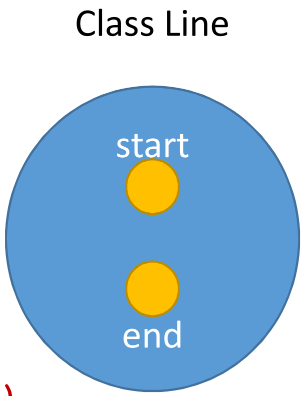
8.1.3.1 Constructor and destructor Chaining in case of embedded objects

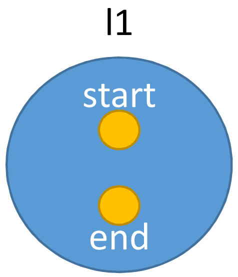
- The question is, which constructor of the embedded object is calling? Where it is not allowed to initialize any attributes inside the class. The answer is: the default constructor of the embedded objects which will be called.
- What if there is no default constructor for its class? Or if I want to let another constructor to be called instead of the default constructor.
- Yes we can, by making redirection to another constructor through the header of the constructor of the container object. In the above example, we make the redirection of the Point objects constructors through the header of the Line constructor, by butting ":" at the end of the header and write the name of embedded object with the parameters to select which constructor you want. As we show below:

```
Line(int x1, int y1, int x2, int y2) : start(x1, y1) , end(x2, y2)
{}
```

8.1.3 Embedded Objects as Strong Association

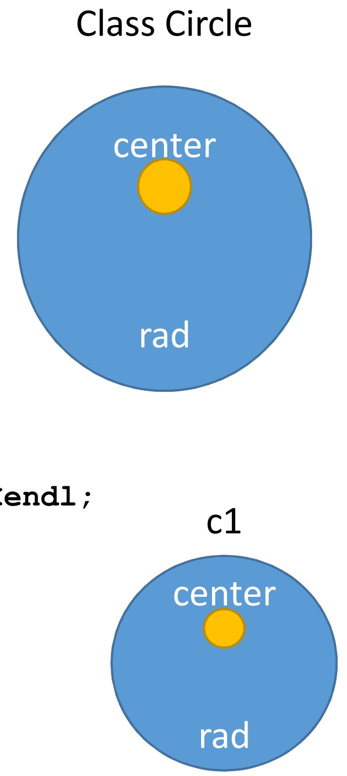
```
class Line
{  private:
       Point start;
       Point end;
  public:
       Line()  : start() , end()
       {      //start.setXY(0,0);end.setXY(0,0);
              cout<<"Line default constructor is calling"<<endl;
       }
       Line(int x1, int y1, int x2, int y2) : start(x1, y1), end(x2, y2)
       {      // start.setXY(x1,y1);    end.setXY(x2,y2);
              cout<<"Line constructor with 4 parameters is calling"<<endl;
       }
       ~Line(){cout<<"Line destructor is calling"<<endl;}
       void print()
       {      cout<<"\nStart:";   start.print();
              cout<<"\nEnd:";     end.print();
       }
};
```





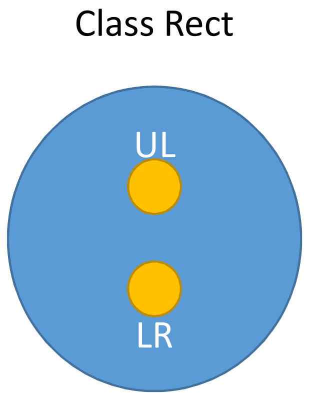
8.1.3 Embedded Objects as Strong Association

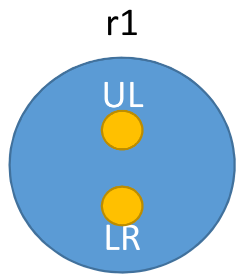
```
class Circle
{  private:
       Point center;
       int rad;
  public:
        Circle() : center() , rad(0)
       {      //center.setXY(0,0);      rad = 0;
              cout<<"Circle default constructor is calling"<<endl;
       }
        Circle(int x1, int y1, int r) : center(x1, y1), rad(r)
       {      //  center.setXY(x1,y1);  rad = 0;
              cout<<"Circle constructor with 3 parameters is calling"<<endl;
       }
       ~Circle(){cout<<"Circle destructor is calling"<<endl;}
       void print()
       {      cout<<"\ncenter:"; center.print();
              cout<<"\nRadius = "<<rad<<endl;
       }
};
```



8.1.3 Embedded Objects as Strong Association

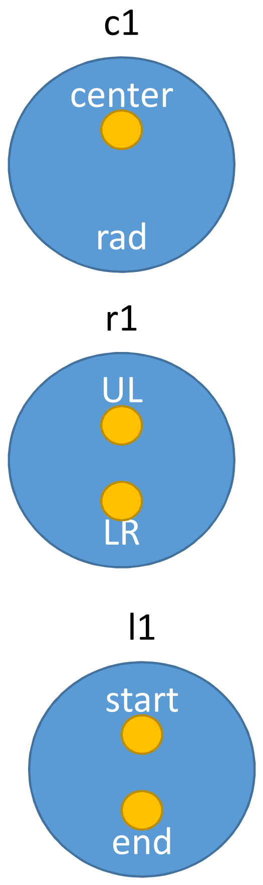
```
class Rect
{  private:
       Point UL;
       Point LR;
  public:
       Rect()  : UL() , LR()
       {      //UL.setXY(0,0);    LR.setXY(0,0);
              cout<<"Rect default constructor is calling"<<endl;
       }
       Rect(int x1, int y1, int x2, int y2) : UL(x1, y1), LR(x2, y2)
       {      // UL.setXY(x1,y1);LR.setXY(x2,y2);
              cout<<"Rect constructor with 4 parameter is calling"<<endl;
       }
       ~Rect(){cout<<"Rect destructor is calling"<<endl;}
       void print()
       {      cout<<"\nUpper Left:";    UL.print();
              cout<<"\nLower Right:";   LR.print();
       }
};
```





8.1.3 Embedded Objects as Strong Association

```
int main()
{
     Circle c1(250,150,100) ;
     Rect r1(10,100,90,350) ;
     Line l1(30,100, 350, 400) ;

     c1.print() ;
     r1.print() ;
     l1.print() ;
     getch() ;
     return 0 ;
}
```



8.2 Collections of objects

- We can deal with objects like the structure, we may crate an array of objects, and we may make dynamic memory allocation with objects.
- We may initialize the array of objects.

8.2.1 Static Allocations : array of objects

- We may declare an array of objects exactly like declare array of structures.

```
Complex carr[10];
```

- The above line is declare an array of 10 Complex objects and each object call the default constructor
- We may call a different constructor for each object as we need:

```
Complex carr[3] = {Complex(2, 4), Complex(), Complex(8)};
```

- If we write constructors to some objects and not to all, the remaining will call the default constructor
- Like any array in C, we may not write the size in array declaration if we make initialization with constructors.

8.2.1 Static Allocations : array of objects

```
int main()
{
    Complex arr[3] = {Complex(2), Complex(), Complex(5,7)};
    for(int i = 0 , i<3 ; i++)
        arr[i].print();
    getch();
    return 0;
}
```

8.2.2 Dynamic Allocation: pointer to object

- We can use a pointer to object to make dynamic memory allocation with number of objects allocated in the heap memory and deal with them like array. The easy of using new operator in C++ make the dynamic memory allocation as piece of cake.

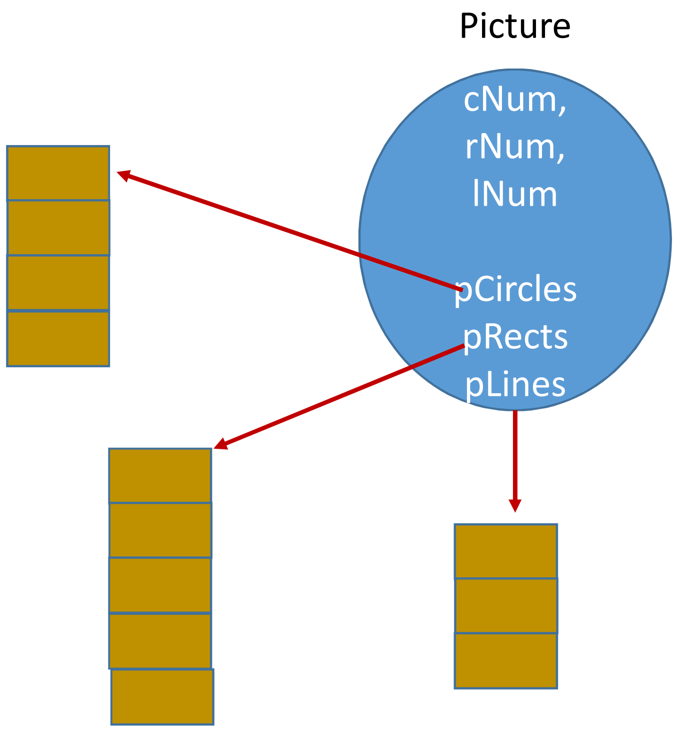
```
Complex * cptr;
cptr = new Complex(2.1, 7.3); // just allocate one Complex

Cptr = new Complex[12]; // allocate 12 complex contiguous
                       // but with default constructor
                       // only
```

8.3 Weak Association: (Aggregation)

- It is a specialized form of association between two or more objects in which the objects have their own life-cycle but there exists an ownership as well. As an example, an employee may belong to multiple departments in the organization. However, if the department is deleted, the employee object wouldn't be destroyed.
- A type of whole-part relationship in which the component parts also exist as individual objects apart from the aggregate.

8.3.1 Example of Picture class with lines circles, and rect

```
class Picture
{
  private :
      int cNum ;
      int rNum ;
      int lNum ;
      Circle *pCircles;
      Rect *pRects;
      Line *pLines;
  public :
      Picture()
      {
            cNum=0;
            rNum=0;
            lNum=0;
            pCircles = NULL ;
            pRects = NULL ;
            pLines = NULL ;
      }
```



8.3.1 Example of Picture class with lines circles, and rect

```
    Picture(int cn, int rn, int ln, Circle *pC, Rect *pR, Line *pL)
    {
        cNum = cn;
        rNum = rn;
        lNum = ln;
        pCircles = pC ;
        pRects = pR ;
        pLines = pL ;
    }

    void setCircles(int, Circle *);
    void setRects(int, Rect *);
    void setLines(int, Line *);
    void print();
};
```

8.3.1 Example of Picture class with lines circles, and rect

```
void Picture::setCircles(int cn, Circle * cptr)
{
     cNum = cn ;
     pCircles = cptr ;
}

void Picture::setRects(int rn, Rect * rptr)
{
     rNum = rn ;
     pRects = rptr ;
}

void Picture::setLines(int ln, Line * lptr)
{
     lNum = ln ;
     pLines = lptr ;
}
```

8.3.1 Example of Picture class with lines circles, and rect

```
void Picture::print()
{
      int i;
      for(i=0; i<cNum ; i++)
      {
            pCircles[i].print();
      }

      for(i=0 ; i<rNum ; i++)
      {
            pRects[i].print();
      }

      for(i=0 ; i<lNum; i++)
      {
            pLines[i].print();
      }
}
```

8.3.1 Example of Picture class with lines circles, and rect

```
int main()
{
    Picture  myPic;
    Circle cArr[3]={Circle(50,50,50), Circle(200,100,100),
                                                 Circle(420,50,30)};
    Rect rArr[2]={Rect(30,40,170,100), Rect(420,50,500,300)};
    Line lArr[2]={Line(420,50,300,300), Line(40,500,500,400)};
    myPic.setCircles(3,cArr) ;
    myPic.setRects(2,rArr) ;
    myPic.setLines(2,lArr) ;
    myPic.print() ;
    getch();
    return 0;
}
```

8.3.1 Example of Picture class with lines circles, and rect

```
int main()
{    Picture   myPic;
     //example on static allocation
     Circle cArr[3]={Circle(50,50,50), Circle(200,100,100),
                                             Circle(420,50,30)};
     //example on static allocation, using temporary objects (on the fly)
     Rect rArr[2] ;
     rArr[0] = Rect(30,40,170,100) ;
     Point myP1(420,50) ;
     Point myP2(500,300) ;
     rArr[1] = Rect(myP1, myP2) ;
```

8.3.1 Example of Picture class with lines circles, and rect

```
    //example on dynamic allocation, using temporary objects (on the fly)
    Line * lArr ;
    lArr = new Line[2] ;
    lArr[0] = Line(Point(420,50) , Point(300,300)) ;
    lArr[1] = Line(40,500,500,400) ;
    myPic.setCircles(3,cArr) ;
    myPic.setRects(2,rArr) ;
    myPic.setLines(2,lArr) ;
    myPic.print() ;
    delete[] lArr ;
return 0;
```

8.4 Aggregation vs Composition

- **Dependency:** Aggregation implies a relationship where the included object **can exist independently** of the container object. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the included object **cannot exist independent** of the container object. Example: Human and heart, heart don't exist separate to a Human
- **Type of Relationship:** Aggregation relation is **"has-a"** and composition is **"part-of"** relation.
- **Type of association:** Composition is a **strong** Association whereas Aggregation is a **weak** Association.

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Assignments

- Example of Point, Rect, Circle, Line as strong association.
- Collections: Simple trials in the main( ) function to create object and array of objects from class Complex
- Association Example: Picture, Point, Rect, Circle, Line

Inheritance

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

Content

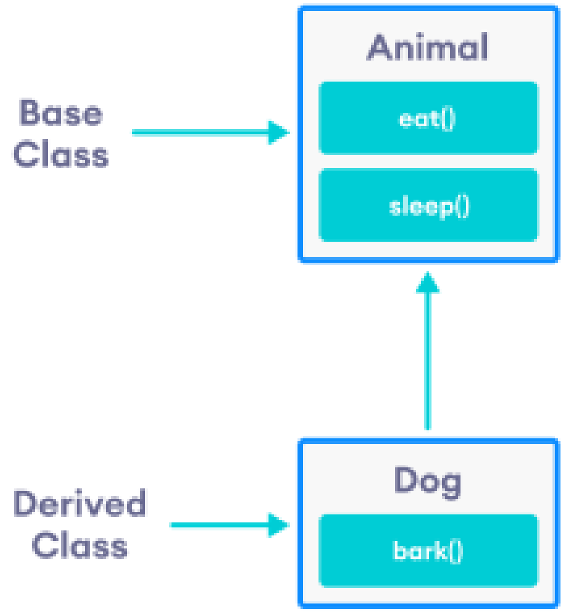
- Inheritance
  - Inheritance is an "is-a" relationship.
  - Advantages of Inheritance
  - Basic concepts of inheriting variables and methods.
  - The "protected" access specifier.
  - Syntax of Inheritance.
- Modes of inheritance
- Overriding.
- Multi-Level Inheritance
- Constructor and Destructor Chaining in case of Inheritance
- Simple Hierarchy Example on overriding and protected: the "calculateSum( )" program.
- The GeoShape class hierarchy: GeoShape, Circle, Triangle, Rectangle, Square).
  - The problem of violating the square constraint.
  - Modification of GeoShape class hierarchy (protected inheritance).

9.1 Inheritance: is-a

- *Inheritance* is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).
- The derived class inherits the features from the base class and can have additional features of its own.
- The capability of a class to derive properties and characteristics from another class is called Inheritance.
- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
  Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.
- It is a relationship between classes where one class is a *parent (super or base)* of another *(Child, subclass, derived)*. It implements *"is-a"* relationships between objects. Inheritance takes advantage of the commonality among objects to reduce complexity.

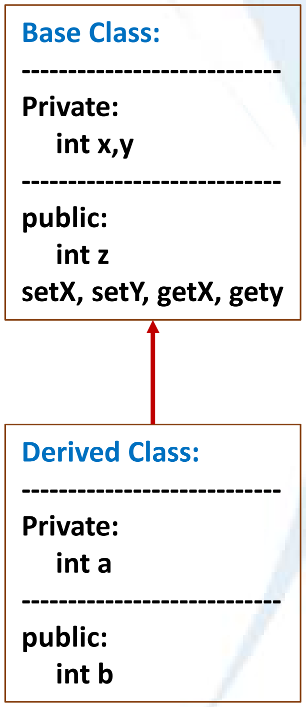
9.1.1 Advantages of Inheritance

- **Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.



9.1.2 Basic concepts of inheriting variables and methods.

```
int main()
{
     Derived d;
     d.b=5;
     d.a=7;         // Not Accessible
     d.z=12;        // It is ok
     d.x=-13; //Not Accessible
     d.setX()12     // It is ok
     cout<<d.getY()     // It is ok
}
```



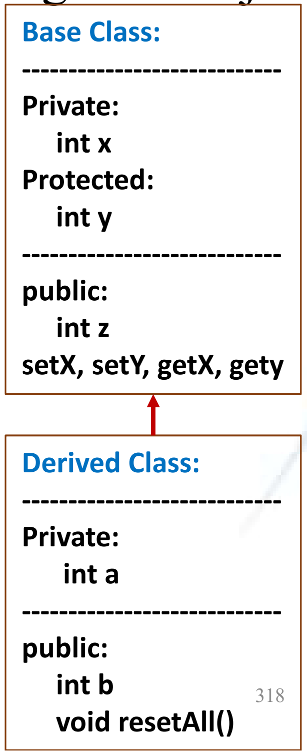
- If a derived class need to access an inherited member which was private in a parent class, it is not allowed, it has to use the inherited public members which access the private members. If a base class want to let its derived classes to access its private members directly, it has to put those members in ***protected*** section.

9.1.3 Protected Section in Class

- The protected members, which declared in protected section, it is like private members in the same class, the different in the case of inheritance where the child class can deal with it directly –by members of the child class not through the objects of the child class- without need to use the public members.

```
void resetAll(){
    x=0;    // Still not accessible use setX()
    y=z=0;  // it is ok
    a=b=0;
}
```

- But still x,y, and a not accessible though using Derived class object.



9.1.4 The Syntax of Inheritance to make a Child Class

```
class Child_class_name : access-mode Parent_class_name
{
    // body of the derived class.
}
```

- Where,
  - **Child_class_name:** It is the name of the derived class.
  - **Access mode:** The access mode specifies whether the features of the base class are publicly inherited, protected inherited, or privately inherited. It can be public, protected, or private.
  - **Parent_class_name:** It is the name of the base class.

9.2 Modes of Inheritance

Parent Sections / Inheritance Mode	Private	Protected	Public
Private	Not Accessible	Private	Private
Protected	Not Accessible	Protected	Protected
Public	Not Accessible	Protected	Public

- The most cases of inheritance is public mode (more than 95%)
- The protected mode is using little.
- The using of private mode is rare, because it is closed the accessibility of the base class after the direct children level –including the public members-

9.3 Polymorphism: Overriding

- If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.
- Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class. A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.
- **Overriding**: A *method* defined in a *superclass* may be overridden by a *method* of the same name defined in a *subclass*. The two *methods* must have the **same name** and **number and types of formal input parameters**.

9.3 Polymorphism: Overriding

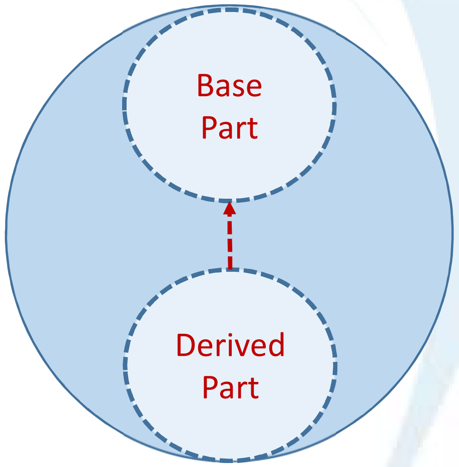
```
class Base {
public:
   void disp(){
      cout<<"Function of Parent Class";
   }
};
class Derived : public Base{
public:
   void disp() {
      cout<<"Function of Child Class";
   }
};
int main() {
   Derived obj = Derived(); // It execute the Base version except if you make
   obj.disp();              // overriding it will print the Derived version.
   obj.Base::disp();        // You may call the function of the base using
                            // the name of the base class with scope operator
}
```

9.4 Multi-Level Inheritance

- In Multilevel Inheritance a derived class can also inherited by another class.
- In object-Oriented Paradigm, classes are organized into a singly rooted tree structure, called the *inheritance hierarchy*.
- **Inheritance Hierarchy**: The relationship between *super classes* and *subclasses* is known as an inheritance hierarchy.
- The words Super Class, Parent Class, and Base Class are represent the same thing.
- The words Sub Class, Child Class, and Derived Class are represent the same thing.

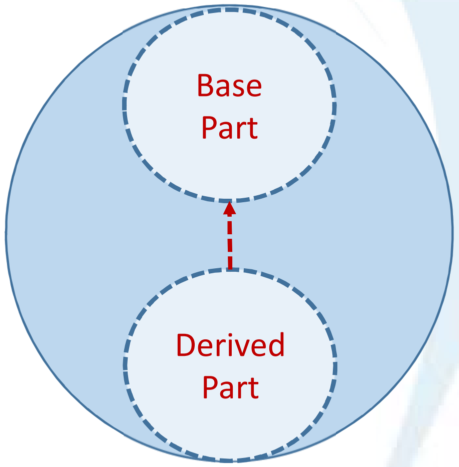
9.5 Constructor and Destructor Chaining in case of Inheritance

- Any object from the derived class consists of two parts: the inherited part from the base class, and the second part is the added members by the derived class.
- Each part has its initial behavior – constructor- the inherited part call the constructor of the base class after it is created, and the other part call the derived constructor after it is created.
- In the inheritance, the constructors of the base and of the derived are calling in sequential, but in which order?
- The base part is created first, then the constructor of the base class will calling first, and then the constructor of the derived class is calling after that.



9.5 Constructor and Destructor Chaining in case of Inheritance

- You may redirect which constructor of the base class is calling, by write as extension to the header of the constructor of the derived class using “:” and select which constructor you want from the base class using the name of the base class.
- The order of the destructor chaining will be the reverse order of the constructor chaining, i.e. the destructor of the derived class will be calling first and then the destructor of the base class will be calling after that



9.6 Simple Example of Inheritance

```
class Base
{
  protected: //first, the student should try it as private
     int a ;
     int b ;
  public:
     Base()
     { a=b=0 ; }

     Base(int n)
     { a=b=n ; }

     Base(int x, int y)
     { a = x ; b = y ; }

     void setA(int x)
     { a = x ; }
```

9.6 Simple Example of Inheritance

```
    void setB(int y)
    { b = y ; }

    int getA()
    { return a ; }

    int getB()
    { return b ; }

    int calculateSum()
    {
        return a + b ;
    }
};
```

9.6 Simple Example of Inheritance

```
class Derived : public Base
{
  private:
    int c ;
  public:
    Derived() : Base()
    { c = 0 ; }

    Derived(int n) : Base(n)
    { c = n ; }

    Derived(int x, int y, int z) : Base(x,y)
    { c = z ; }
```

9.6 Simple Example of Inheritance

```
    void setC(int z)
    { c = z ; }

    int getC()
    { return c ; }

    int calculateSum()  //overriding
    {
        return a + b + c ;  // only if "a" and "b" were protected,
                            // or use getters if it is private
        //  other implementation ideas:
        //  return Base::calculateSum() + c ;
    }
};
```

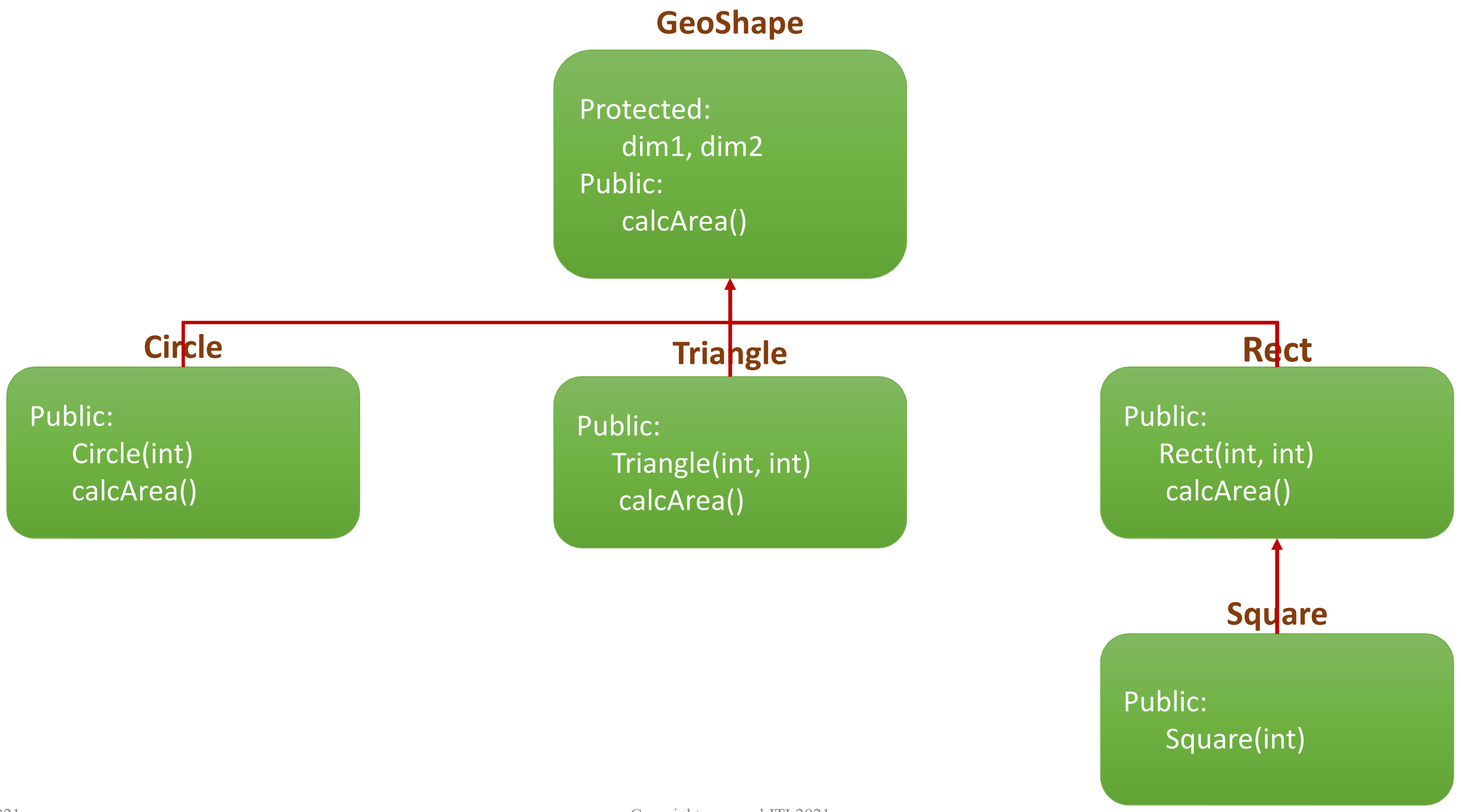
9.6 Simple Example of Inheritance

```
int main()
{
    clrscr() ;
    Base b(5,4) ;
    cout<<b.calculateSum()<<endl ;
    Derived obj1 ;
    obj1.setA(3) ;
    obj1.setB(7) ;
    obj1.setC(1) ;
    Derived obj2(20) ;
```

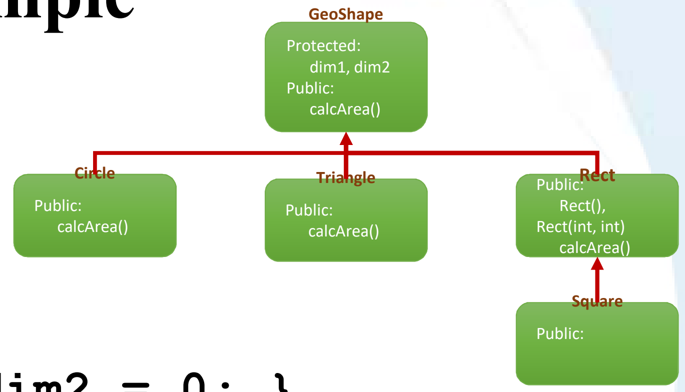
9.6 Simple Example of Inheritance

```
    Derived obj3(4,5,6) ;
    cout<<"obj1: "<<obj1.calculateSum()<<endl ; // =11
    cout<<"obj2: "<<obj2.calculateSum()<<endl ; // =60
    cout<<"obj3: "<<obj3.calculateSum()<<endl ; // =15
cout<<"obj1: "<<obj1.Base::calculateSum()<<endl ; //only =10
    getch() ;

return 0 ;
}
```

9.7 The GeoShape Class Hierarchy Example



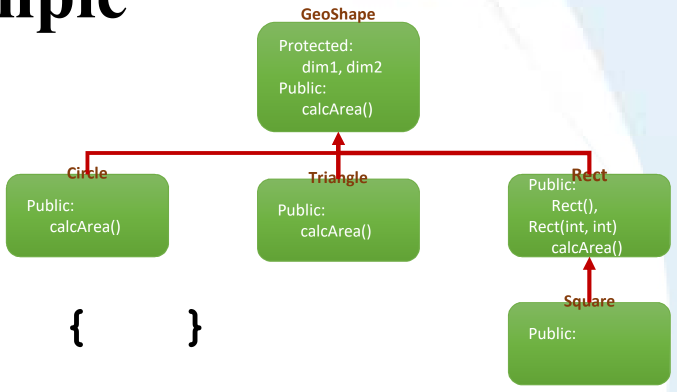
9.7 The GeoShape Class Hierarchy Example



```
class GeoShape
{  protected:
      float dim1, dim2;
    public:
      GeoShape()                  { dim1 = dim2 = 0; }
      GeoShape(float x)           { dim1 = dim2 = x; }
      GeoShape(float x, float y)  { dim1 = x;   dim2 = y;   }
      void setDim1(float x)       { dim1 = x; }
      void setDim2(float x)       { dim2 = x; }
      float getDim1()             { return dim1; }
      float getDim2()             { return dim2; }
      float calcArea()            { return 0.0;   }
};
```

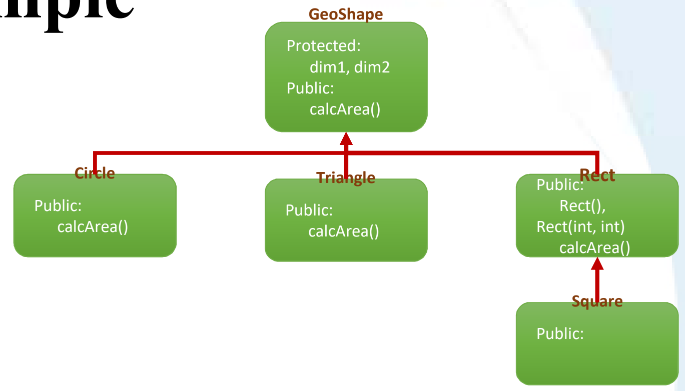
9.7 The GeoShape Class Hierarchy Example

```
class Rect: public GeoShape
{  public:

     Rect(float x, float y) : GeoShape(x, y)  {    }
     float calcArea()
     {
          return dim1 * dim2;
     }
};
class Square: public Rect
{  public:
     Square(float x) : Rect(x, x) {    }
};
```



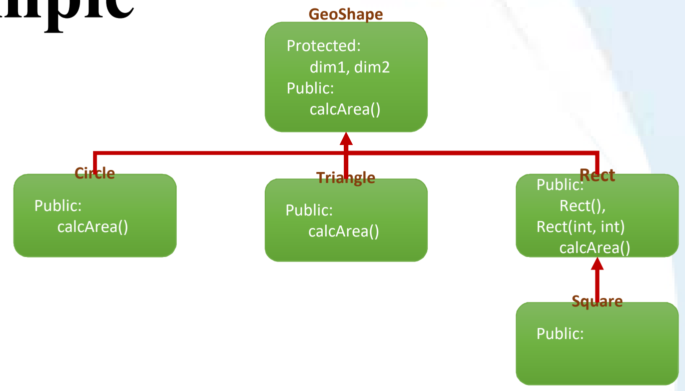
9.7 The GeoShape Class Hierarchy Example

```
class Triangle : public GeoShape
{  public:
     Triangle(float b, float h):GeoShape(b, h){ }
     float calcArea()
     {
          return 0.5 * dim1 * dim2;
     }
};
class Circle : public GeoShape
{  public:
     Circle(float r) : GeoShape(r)   {    }
     float calcArea()
     {                                          //it's ok since dim1 equals to dim2.
          return 22.0/7 * dim1 * dim2; //you may write:  22.0/7*dim1*dim1.
     }
};
```



9.7 The GeoShape Class Hierarchy Example

```
int main()
{   Triangle myT(20, 10);
    cout << myT.calcArea() << endl;
    Circle myC(7);
    cout << myC.calcArea() << endl;
    Rect myR(2, 5);
    cout << myR.calcArea() << endl;
    Square myS(5);
    cout << myS.calcArea() << endl;
    //What happened if you try:
    mySetDim2(4) ; //Violating the Square Constraint
    myC.setDim2(3) ; //Violating the Circle Constraint
    getch() ;
    return 0;
}
```

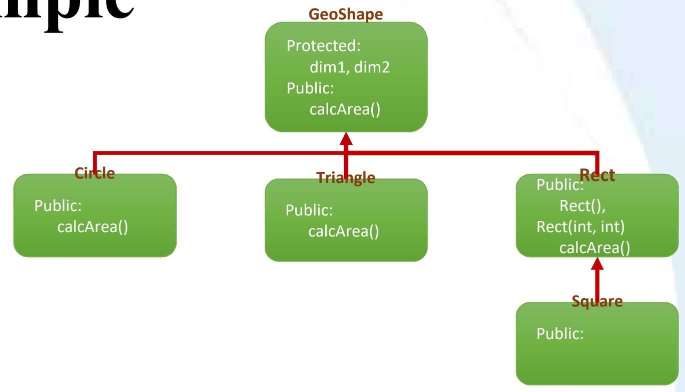


9.7 The GeoShape Class Hierarchy Example

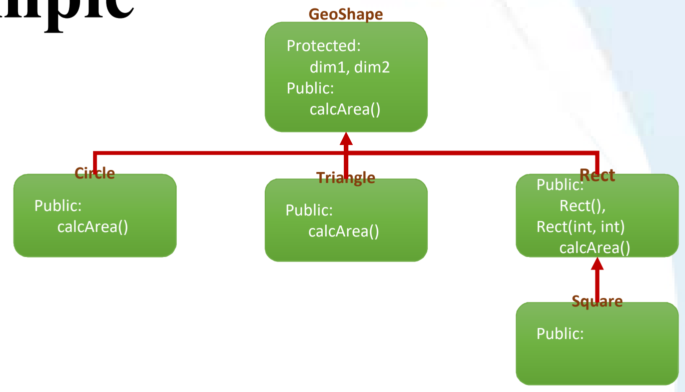
- In the above example there are cases *violate* the constraint of the Square logic; the two dimensions are the same equal to the *side length*. And so the constraint of the circle logic; where the two dimensions are the same equal to the *radius*.
- This happened because the public inheritance of Square class from Rect class, where the setters *setDim1* and *setDim2* is accessible inside the Square class and so through an object from the Square class. How this logical problem can be solving?
- You may ***override*** the two setters *setDim1* and *setDim2* inside the Square class to do the same thing: change in the two dimensions together. But this solution may confusing the user of the class.
- Other better solution, make the inheritance mode to be ***protected*** mode when you inherit the Square class from the Rect class, and make a public members with name *setSide(int)* which set the two dimensions with the same value. But you have to override the *calcArea()* in class Square.
- Do the same thing with the Circle class.

9.7 The GeoShape Class Hierarchy Example

```
class Square: protected Rect
{  public:
     Square(float x) : Rect(x, x)  {   }
     void setSide(int length)
     {
         dim1 = dim2 = length;
     }
     float calcArea()
     {
         return dim1 * dim2;
     }
};
```



9.7 The GeoShape Class Hierarchy Example



```
class Circle : protected GeoShape
{   public:
      Circle(float r) : GeoShape(r)
      {    }
      void setRadius(float rad)
      {
          dim1 = dim2 = rad;
      }
      float calcArea()
      {                                   //it's ok since dim1 equals to dim2.
          return 22.0/7*dim1*dim2;//you may write:  22.0/7*dim1*dim1.
      }
};
```

Lab Exercise

Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Assignments

- Example of: Base, Derived, and calculateSum( ).
- GeoShape Example:
    - Try with public inheritance (try to violate the square and circle constraints by calling the inherited setter methods).
    - Try with protected inheritance (not able to violate the square and circle constraints).
    - Add the new special setter functions to class square and circle.

Inheritance II

Java™ Education
and Technology Services

Invest In Yourself,
Develop Your Career

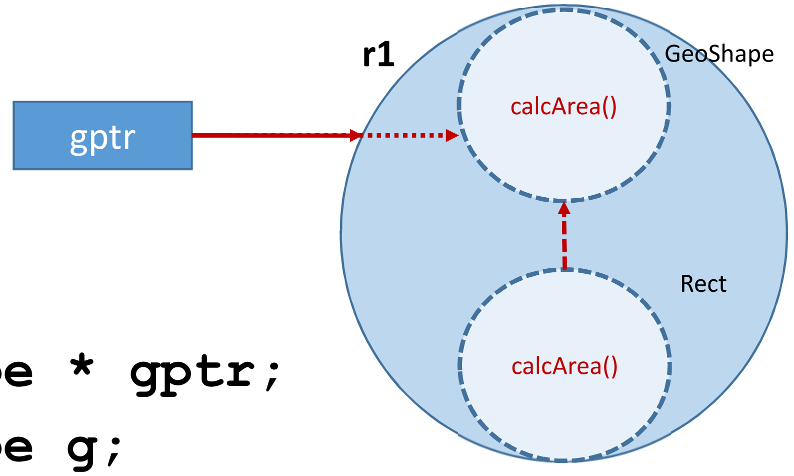
Content

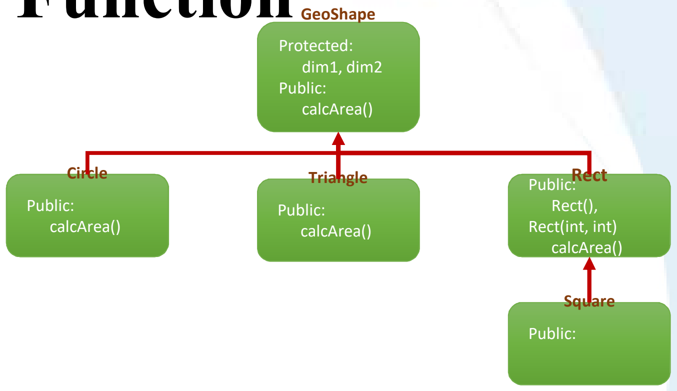
- Dynamic Binding and Virtual Function
- Example of Dynamic Binding: sumOfAreas(, ,) as stand alone function
- Pure Virtual Functions (Abstract Method), and Abstract Class Vs Concrete class
- Multiple Inheritance and its problems and its’ solutions: Virtual Base Class
- Templates in C++: Stack example

10.1 Dynamic Binding and Virtual Function

- Dynamic binding also called dynamic dispatch is the process of linking method call to a specific sequence of code at run-time. It means that the code to be executed for a specific method call is not known until run-time.
- The way of perform the dynamic binding using the ***Virtual Function***.
- A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function. It is the mechanism of ***Dynamic Binding*** concept.
- For example: the function calcArea() which is defined as a members in class GeoShape, the expectation to be overridden is near to be 100%, so we have to make it virtual, by add the reserved word ***"virtual"*** before the returned data type of the header of the function as a function modifier.

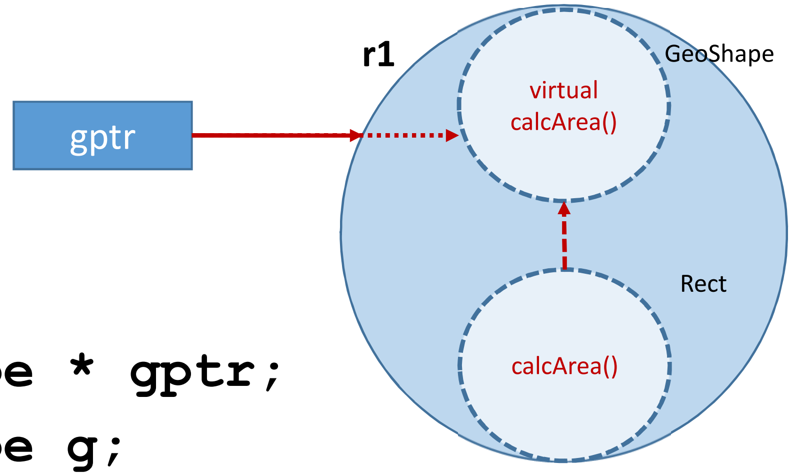
10.1 Dynamic (late) Binding and Virtual Function

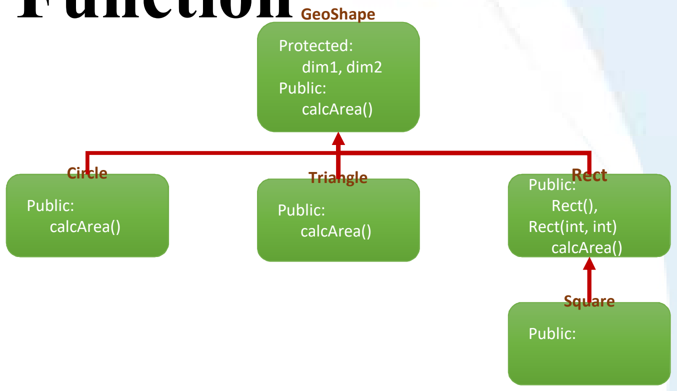




```
Geoshape * gptr;
Geoshape g;
Rect r1;
gptr = & g;
cout<<gptr->calcArea();
gptr = & r1;
cout<<gptr->calcArea(); //the GeoShape version is called
```

10.1 Dynamic (late) Binding and Virtual Function





```
Geoshape * gptr;
Geoshape g;
Rect r1;
gptr = & g;
cout<<gptr->calcArea();
gptr = & r1;
cout<<gptr->calcArea(); //the Rect version if it is virtual
```

10.1 Dynamic (late) Binding and Virtual Function

```
class Rect: public GeoShape
{  public:

     Rect(float x, float y) : GeoShape(x, y)   {   }
     virtual float calcArea()
     {
          return dim1 * dim2;
     }
};
```

10.2 Example of Dynamic Binding: sumAreas(, ,)

- If we need a function to make the sum of any three objects from type of GeoShape – from all of its children-, it is an example of using of the dynamic binding.
- This function will take three object by their address, the calling it will be by address and it will send to pointer to GeoShape.

```
float sumAreas(GeoShape * p1, GeoShape * p1, GeoShape * p3)
{
    return p1->calcArea()+p2->calcArea()+p3->calcArea();
}
```

- This function will execute on any three objects from type GeoShape; i.e. any objects from GeoShape or any objects from children of GeoShape.
- When you call this function, you may send three object from Rect, or send 1 from Circle and 2 from Square, and so on, the version pf the calcArea will be dedicated the run time according to the type of the object.
- Note: Make this function as stand alone function.

10.2 Example of Dynamic Binding: sumAreas(, ,)

```
int main()
{
    Rect r1(1,2,3,4), r2(5,10,15,20);
    Square s1(10), s2(20), s3(30);
    Circle c1(30);
    Triangle t1(10,20), t2(20, 40);
    cout<<sumAreas(&s1, &s2, &s3);
    cout<<sumAreas(&t1, &c2, &r2);
    cout<<sumAreas(&r1, &s2, &t2);
    getch();
    return 0;
}
```

10.2 Example of Dynamic Binding: sumAreas(, ,)

• The version of the function using references:

```
float sumAreas(GeoShape & r1, GeoShape & r1, GeoShape & r3)
{
    return r1.calcArea()+r2.calcArea()+r3.calcArea();
}
```

10.2 Example of Dynamic Binding: sumAreas(, ,)

```
int main()
{
    Rect r1(1,2,3,4), r2(5,10,15,20);
    Square s1(10), s2(20), s3(30);
    Circle c1(30);
    Triangle t1(10,20), t2(20, 40);
    cout<<sumAreas(s1, s2, s3);
    cout<<sumAreas(t1, c2, r2);
    cout<<sumAreas(r1, s2, t2);
    getch();
    return 0;
}
```

10.3 Pure Virtual Function and Abstract Class

- **Abstract Method (Pure Virtual Function)** is a *method* that is declared, but contains no implementation, without body. It is needed to be overridden by the subclasses for standardization or generalization.
- **Abstract Class** is distinguished by the fact that you may not directly construct objects from it. An abstract class may have one or more *abstract methods*. Abstract classes may not be instantiated, and require *subclasses* to provide implementations for the abstract methods. It is needed for standardization or generalization.
- **Concrete Class**: Any class can be initiating objects directly from it, it is not abstract class.
- For Example: the calcArea method of the GeoShape class it must be abstract method, so the class GeoShape will be an abstract class.
- The Abstract method (or pure virtual function in C++) conists of header only without body, the header start with word "***virtual***" and end with "***= 0***".

10.3 Pure Virtual Function and Abstract Class

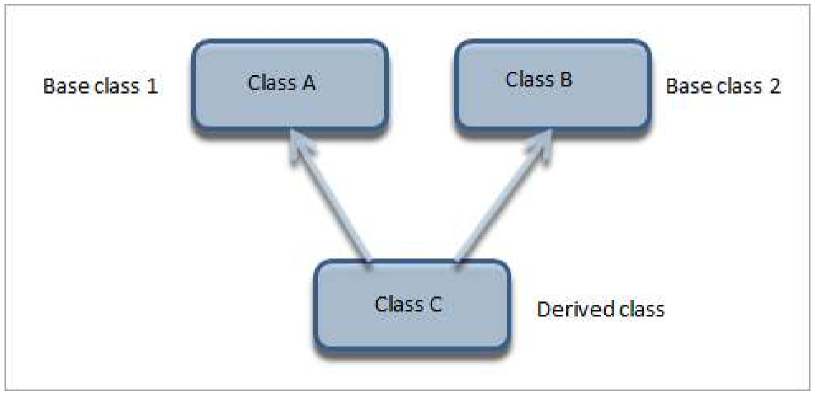
```
class GeoShape
{
    public:
    …
    virtual float calcArea() = 0; // pure virtual function
    …


};
```

10.3 Pure Virtual Function and Abstract Class

```
int main(){
    GeoShape gobj;      // Compilation error can not create
                        // an object from abstract class
    GeoShape * gptr;    // It is allowed to point to any
                        // object from any children classes
```

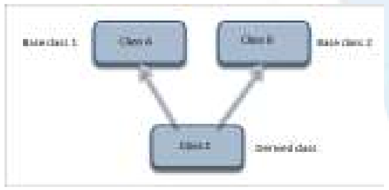
10.4 Multiple Inheritance

- The ability of a class to extend more than one class; i.e. A *class* can inherit characteristics and features from more than one parent class.



- Multiple inheritance is a type of inheritance in which a class derives from more than one classes. As shown in the above diagram, class C is a subclass that has class A and class B as its parent.
- In a real-life scenario, a child inherits from its father and mother. This can be considered as an example of multiple inheritance.

10.4 Multiple Inheritance

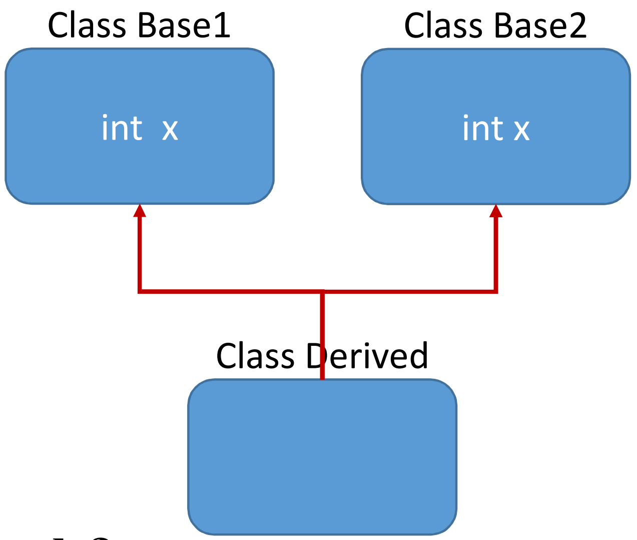


```
class Base1
{}
class Base2
{}
class Derived: public Base1, Base2
{}
```

- All the rules of the inheritance is applied.
- But there are some problems may be faced with multiple inheritance, like if we have two members in the two base classes have the same name, it is solved by using the scope operator (::) with the name of one of the base classes to select which one you want to access. But what if this variable was inherited before to the two base classes from common parent for them?. It is called the ***Diamond Problem***.

10.4.1 Duplicate members name from different base classes

```
Derived d;
d.X = 10; // ambiguity which x?
          // to solve that using ::
d.Base1::x = 10;
d.Base2::x = 20;
…
```



- But what if this member in the two base classes is the same one; has the same original from above levels?
- That what we call, “Diamond Problem”

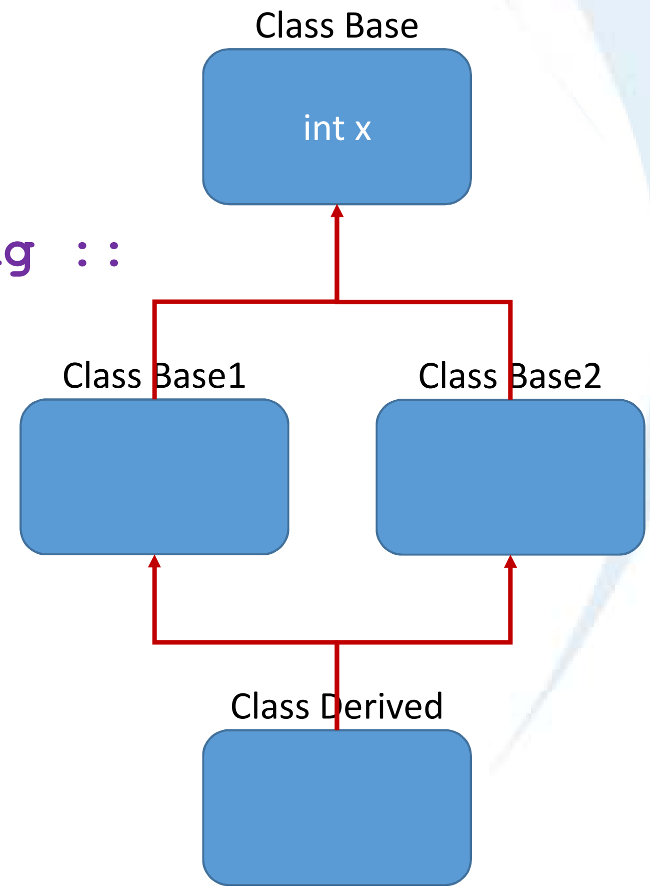
10.4.2 Diamond Problem

```
Derived d;
d.X = 10;               // ambiguity which x?
                        // to solve that using ::
d.Base1::x = 10;        // Not Meaningful
d.Base2::x = 20;        // Not Meaningful
```

…

- The solve by make Base1 and Base2 inherit from class base virtual, how?

```
class Base1: virtual public Base {}
class Base2: public virtual Base {}
```



10.5 Templates in C++ (Generic)

- Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.
- Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.
- The concept of templates can be used in two different ways:
  - Function Templates
  - Class Templates
- C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.

10.5.1 Function Template

- We write a generic function that can be used for different data types. It is similar to a normal function, with one key difference; a single function template can work with different data types at once but, a single normal function can only work with one set of data types.
- Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.
- However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.
- The general form of a template function definition is shown here:

```
template <class type>
return-type function-name(parameter list) {
       // body of function
}
```

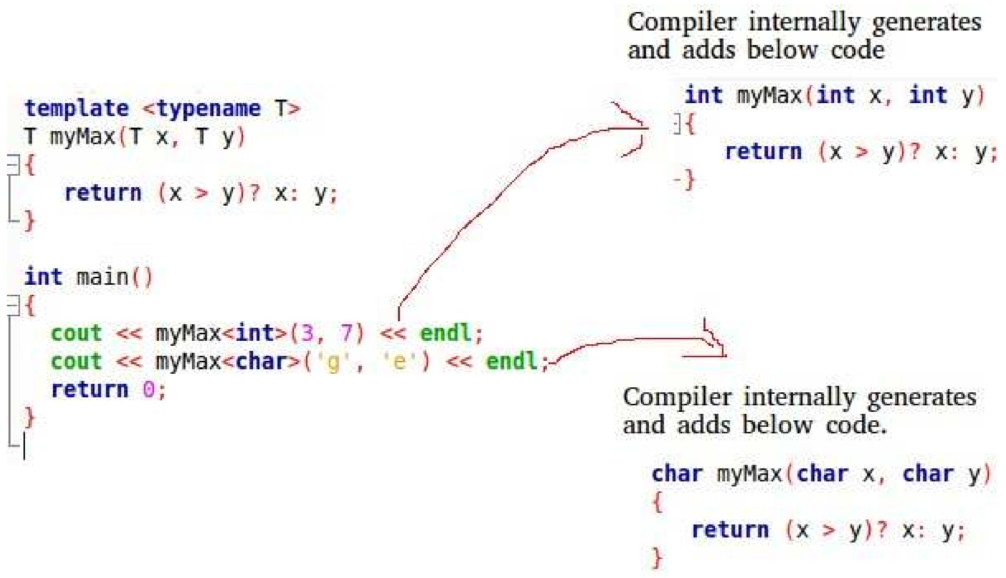
10.5.2 Class Template

- You can create class templates for generic class operations. Sometimes, you need a class implementation that is same for all classes, only the data types used are different.
- Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.
- This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.
- However, class templates make it easy to reuse the same code for all data types.
- The general form of a generic class declaration is shown here:

```
template <class type>
class class-name {
    .
    .
}
```

10.5.3 How templates work?

- Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

10.5.4 Stack Class as Generic Class

```
template <class T>
class Stack
{  private:
      int top, size;
       T *ptr;
      static int counter ;
  public:
      Stack()
      {     top = 0 ;
            size = 10;
            ptr = new T[size];
            counter++ ;
      }
      Stack(int n)
      {     top = 0;
            size = n;
            ptr = new T[size];
            counter++ ;
      }
```

10.5.4 Stack Class as Generic Class

```
    ~Stack()
    {
        delete[] ptr;
        counter-- ;
    }
    static int getCounter()
    {
        return counter ;
    }
    Stack(Stack & s) ;
    int push(T);
    int pop(T & n);
    Stack& operator= (Stack& s);
    friend void viewContent(Stack s) ;
};
```

10.5.4 Stack Class as Generic Class

```
//static variable initialization
template <class T>
int Stack< T >::counter = 0 ;

template <class T >
Stack<T>::Stack(Stack<T> & myStk)
{
    top = myStk.top;
    size = myStk.size;
    ptr = new T[size];
    for (int i=0 ; i<top ; i++)
    {
        ptr[i] = myStk.ptr[i];
    }
    counter++ ;
}
```

10.5.4 Stack Class as Generic Class

```
template <class T>
int Stack<T>::push(T n)
{
    if (isFull())
    {
        return 0;
    }
    else
    {
        ptr[top] = n;
        top++;
        return 1;
    }
}
```

10.5.4 Stack Class as Generic Class

```
template <class T>
int Stack<T>::pop(T & data)
{
    if (isEmpty())
    {
        return 0;
    }
    else
    {
        top--;
        data = ptr[top];
        return 1 ;
    }
}
```

10.5.4 Stack Class as Generic Class

```
template <class T>
Stack<T>& Stack<T>::operator= (Stack<T>& myS)
{
    if(ptr)    delete[] ptr;
    top = myS.top;
    size = myS.size;
    ptr = new T[size];
    for (int i = 0; i < top; i++)
    {
        ptr[i] = myS.ptr[i];
    }
    return *this;
}
```

10.5.4 Stack Class as Generic Class

```
template <class T>
void viewContent(Stack<T> myS)
{
    for(int i=0 ; i<myS.top ; i++)
    {
        cout<<myS.ptr[i]<<endl ;
    }
}
```

10.5.4 Stack Class as Generic Class

```
int main()
{   int n;
    clrscr();
    Stack<int> s1(5);
cout<<"\nNo of Int Stacks is:"<<Stack<int>::getCounter();
    s1.push(10);
    s1.push(3);
    s1.push(2);
    s1.pop(n);
    cout << "\n1st integer: " <<n;
    s1.pop(n);
    cout << "\n2nd integer: " <<n;
```

10.5.4 Stack Class as Generic Class

```
    Stack<char> s2;
    char nc;
cout <<"\nNo of Char Stacks is:"<<Stack<char>::getCounter();
    s2.push('q');
    s2.push('r');
    s2.push('s');
    viewContent(s2) ;
    s2.pop(cn);
    cout << "\n1st character: " <<nc;
    s2.pop(nc);
    cout << "\n2nd character: " <<nc;
    getch() ;
return 0;
}
```

10.6 Introduction to UML

- **What is UML?**
  - The Unified Modeling Language (UML) is the standard modeling language for software and systems development.
  - Modeling helps you to focus on, capture, document, and communicate the important aspects of your system’s design.
  - A modeling language can be made up of pseudo-code, actual code, pictures, diagrams, or long passages of description; in fact, it’s pretty much anything that helps you describe your system.
  - The elements that make up a modeling language are called its *notation* (a way of expressing the model).
  - A modeling language can be anything that contains a notation and a description of what that notation means.

10.6 Introduction to UML

- **UML Advantages**
  - **It's a formal language**: Each element of the language has a strongly defined meaning, so you can be confident that when you model a particular facet of your system it will not be misunderstood.
  - **It's concise**: The entire language is made up of simple and straightforward notation.
  - **It's comprehensive**: It describes all important aspects of a system.
  - **It's scalable**: Where needed, the language is formal enough to handle massive system modeling projects, but it also scales down to small projects.
  - **It's built on lessons learned**: UML is the result of best practices in the object-oriented community during the decades.
  - **It's the standard**: UML is controlled by an open standards group with active contributions from a worldwide group of vendors and academics. (*http://www.omg.org*)

10.6 Introduction to UML

- UML Diagrams

Diagram Type	What Can be modeled?	Originally introduced by UML 1.x or UML 2.0
Use Case	Interactions between your system and users or other external systems. Also helpful in mapping requirements to your systems.	UML 1.x
Activity	Sequential and parallel activities within your system.	UML 1.x
Class	Classes, types, interfaces, and the relationships between them.	UML 1.x
Object	Object instances of the classes defined in class diagrams in configurations that are important to your system.	Informally UML 1.x
Sequence	Interactions between objects where the order of the interactions is important.	UML 1.x
Communication	The ways in which objects interact and the connections that are needed to support that interaction.	Renamed from UML 1.x's collaboration diagrams

10.6 Introduction to UML

- UML Diagrams

Diagram Type	What Can be modeled?	Originally introduced by UML 1.x or UML 2.0
Timing	Interactions between objects where timing is an important concern.	UML 2.0
Interaction Overview	Used to collect sequence, communication, and timing diagrams together to capture an important interaction that occurs within your system.	UML 2.0
Composite Structure	The internals of a class or component, and can describe class relationships within a given context.	UML 2.0
Component	Important components within your system and the interfaces they use to interact with each other.	UML 1.x, but takes on a new meaning in UML 2.0
Package	The hierarchical organization of groups of classes and components.	UML 2.0

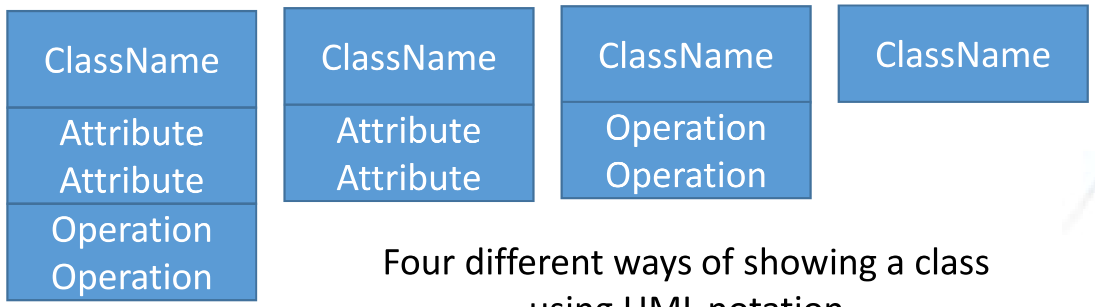
10.6 Introduction to UML

- UML Diagrams

Diagram Type	What Can be modeled?	Originally introduced by UML 1.x or UML 2.0
State Machine	The state of an object throughout its lifetime and the events that can change that state.	UML 1.x
Deployment	How your system is finally deployed in a given real world situation.	UML 1.x

10.6 Introduction to UML

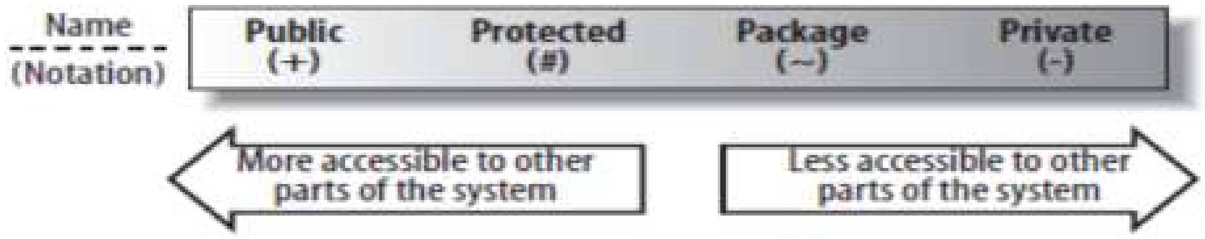
- **Classes in UML**
  - A class in UML is drawn as a rectangle split into up to three sections.
  - The top section contains the name of the class.
  - The middle section contains the attributes or information that the class contains.
  - The final section contains the operations that represent the behavior that the class exhibits.



Four different ways of showing a class using UML notation

10.6 Introduction to UML

- **Visibility**
  - Visibility describes how a class reveals its operations and data to other classes.
  - Once visibility characteristics are applied, you can control access to attributes, operations, and even entire classes to effectively enforce encapsulation.
  - There are four different types of visibility that can be applied to the elements of a UML model:


Name
(Notation)
Public
(+)
Protected
(#)
Package
(~)
Private
(-)
More accessible to other parts of the system
Less accessible to other parts of the system

10.6 Introduction to UML

- **Public Visibility**
  - Public visibility is specified using the plus (+) symbol before the associated attribute or operation.
  - Declare an attribute or operation public if you want it to be accessible directly by any other class.
  - The collection of attributes and operations that are declared public on a class create that class's public interface.
  - The public interface of a class consists of the attributes and operations that can be accessed and used by other classes.
  - This means the public interface is the part of your class that other classes will depend on the most.

10.6 Introduction to UML

- **Protected Visibility**
  - Protected attributes and operations are specified using the hash (#) symbol and are more visible to the rest of your system than private attributes and operations, but are less visible than public.
  - Declared protected elements on classes can be accessed by methods that are part of your class and also by methods that are declared on any class that inherits from your class.

10.6 Introduction to UML

- **Package Visibility**
  - Package visibility, specified with a tilde (~), when applied to attributes and operations, sits in between protected and private.
  - Packages are the key factor in determining which classes can see an attribute or operation that is declared with package visibility.
  - The rule is fairly simple:

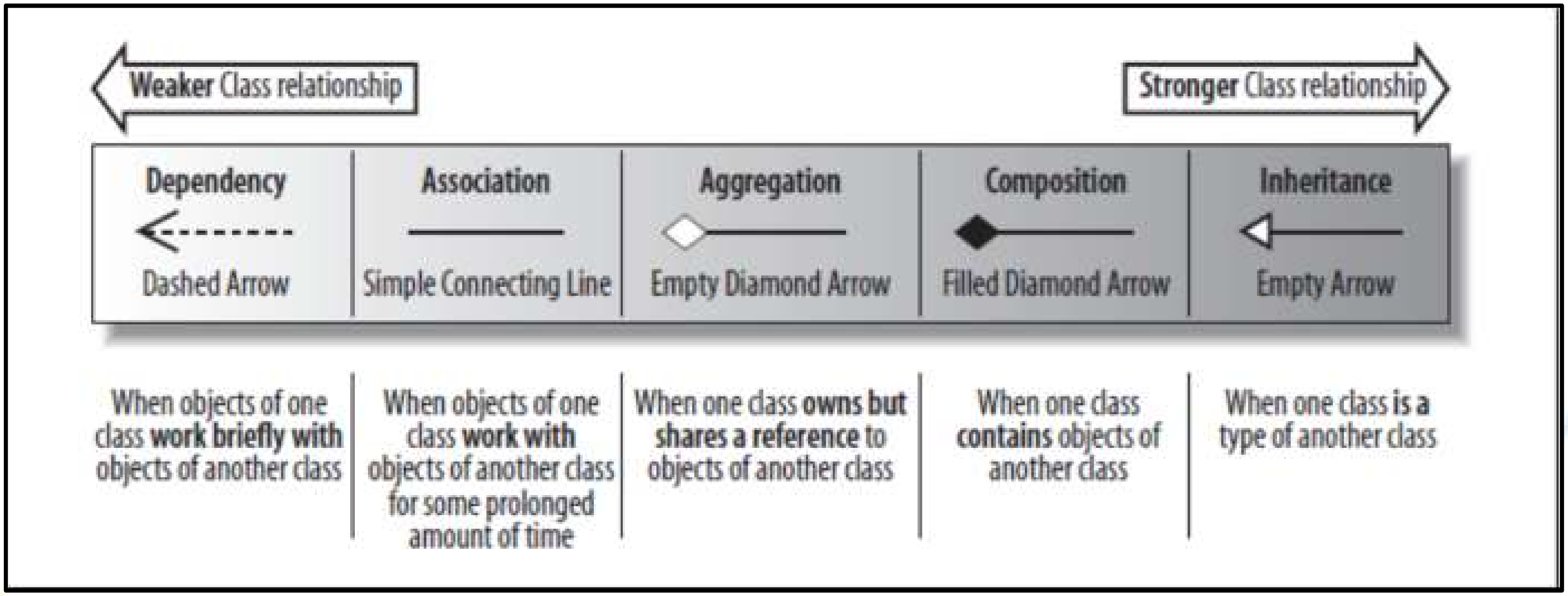
if you add an attribute or operation that is declared with package visibility to your class, then any class in the same package can directly access that attribute or operation

10.6 Introduction to UML

- **Private Visibility**
  - Private visibility is the most tightly constrained type of visibility classification, and it is shown by adding a minus (-) symbol before the attribute or operation.
  - Only the class that contains the private element can see or work with the data stored in a private attribute or make a call to a private operation.

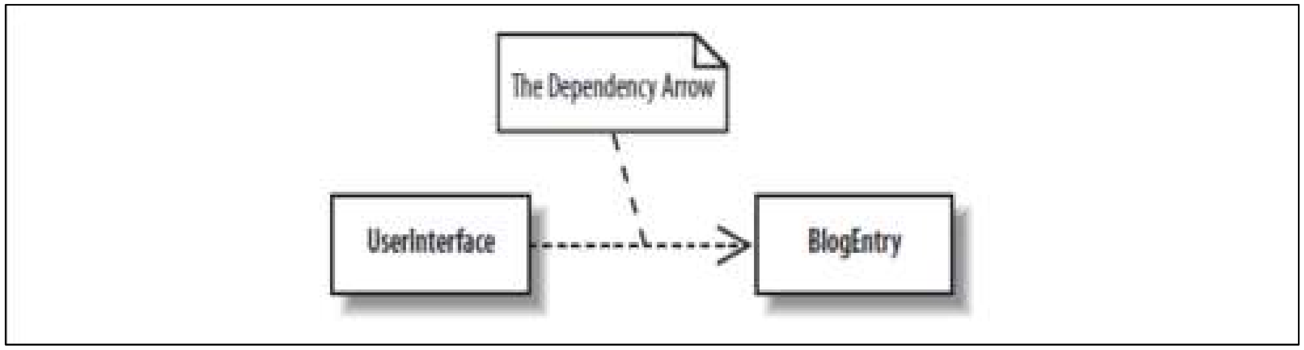
10.6 Introduction to UML

- **Class Relationships**


Weaker Class relationship
Stronger Class relationship
Dependency
Dashed Arrow
When objects of one class work briefly with objects of another class
Association
Simple Connecting Line
When objects of one class work with objects of another class for some prolonged amount of time
Aggregation
Empty Diamond Arrow
When one class owns but shares a reference to objects of another class
Composition
Filled Diamond Arrow
When one class contains objects of another class
Inheritance
Empty Arrow
When one class is a type of another class

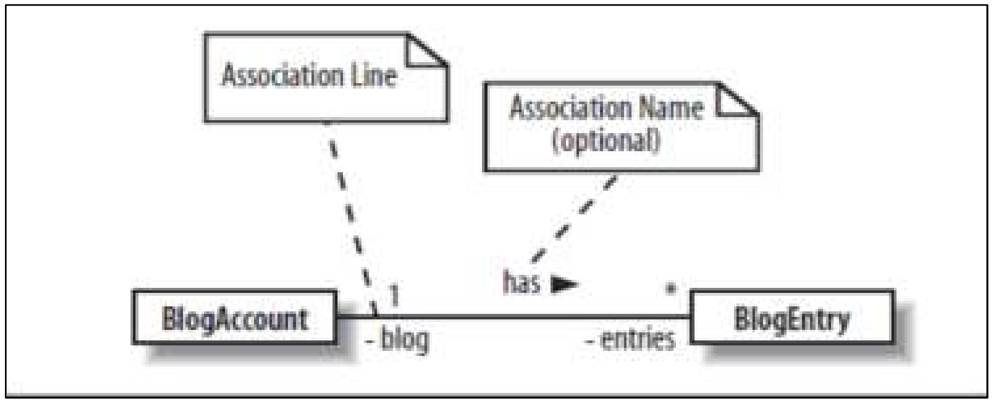
10.6 Introduction to UML

- **Dependency**
  - The dependency relationship is often used when you have a class that is providing a set of general-purpose utility functions, such as in Java's regular expression (java.util.regex) and mathematics (java.math) packages.
  - A dependency between two classes declares that a class needs to know about another class to use objects of that class.



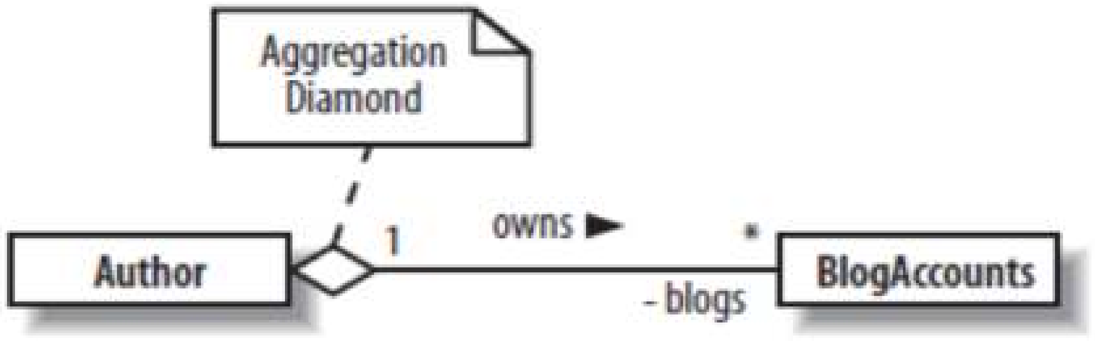
10.6 Introduction to UML

- **Association**
  - Although dependency simply allows one class to use objects of another class, association means that a class will actually contain a reference to an object, or objects, of the other class in the form of an attribute.
  - If you find yourself saying that a class works with an object of another class, then the relationship between those classes is a great candidate for association rather than just a dependency.



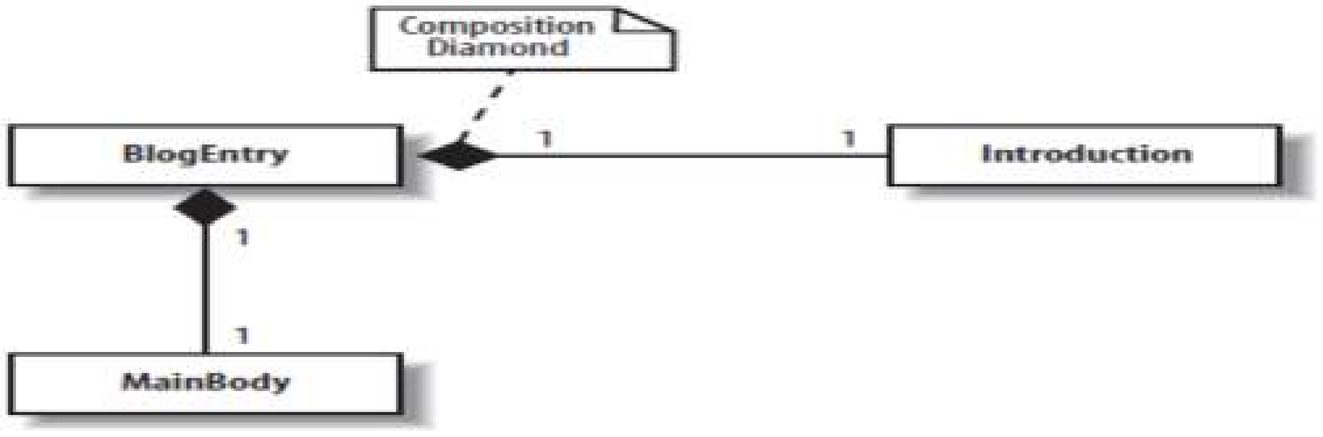
10.6 Introduction to UML

- **Aggregation**
  - Aggregation is really just a stronger version of association and is used to indicate that a class actually owns but may share objects of another class.
  - Aggregation is shown by using an empty diamond arrowhead next to the owning class, as shown in the figure.



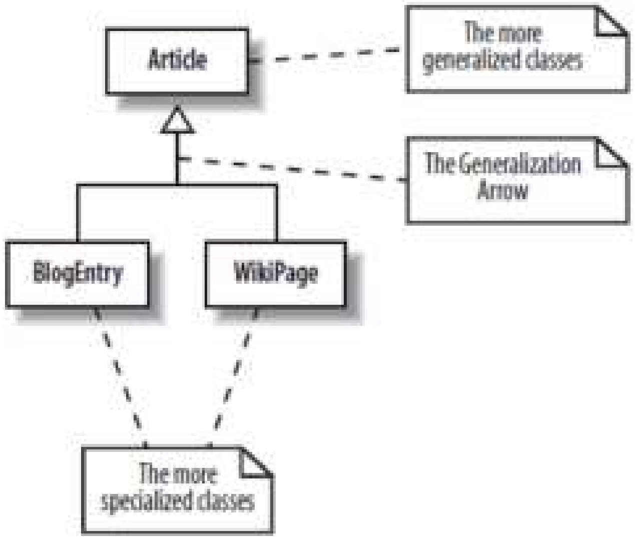
10.6 Introduction to UML

- **Composition**
  - Composition is an even stronger relationship than aggregation, although they work in very similar ways.
  - Composition is shown using a closed, or filled, diamond arrowhead, as shown in the figure:

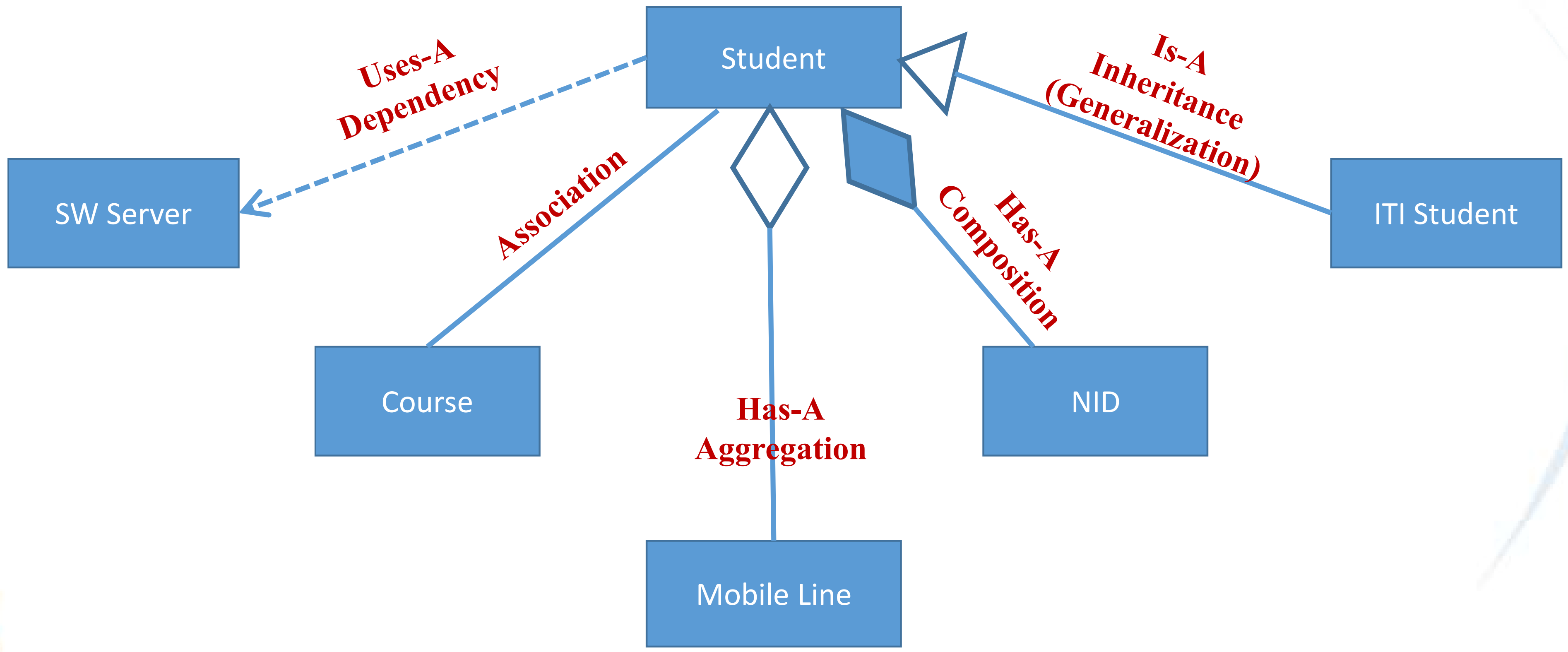


10.6 Introduction to UML

- **Inheritance (Generalization)**
  - Generalization is the strongest form of class relationship because it creates a tight coupling between classes.
  - The child (specialized) class inherits all of the attributes and methods that are declared in the parent (generalized) class and may add operations and attributes that are only applicable in child (specialized) class.
  - In UML, the generalization arrow is used to show that a class is a type of another class, as shown in the figure:



10.6 Introduction to UML



Lab Exercise

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

Assignments

- Continue the GeoShape Example:
  - Make the function: "calculateArea( )", a pure virtual function, and make necessary changes to other classes.
  - Write the standalone function: "sumOfAreas(~, ~, ~), which takes 3 parameters as pointers from type GeoShape.
  - Not able to create objects of the abstract class: GeoShape.
- Stack class with Templates.

With My Best Wishes

Ahmed Loutfy

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career