BASIC C++ Notes

DAY 01

- **Main As Application Entry Point Function**
- Can any function be an entry point function?
  * No. By default, main is the entry point function for console application.
  * And WinMain is the entry point function for Windows Application.
  * If we do not mention entry point function as expected in the project, we get linking error as below:
  * error LNK2019: unresolved external symbol main referenced in function "int _cdecl invoke_main(void)" (?invoke_main@@YAHXZ)

- **Null Statement**

-what is null statement?
-Its an expression statement with the missing expression.

- **Constant**

- C++ classifies 'M', 'A', 'F', '\t', '\n', '\a' as narrow characters or char.
- C++ classifies L'M', L'G' as wide characters or wchar_t.
- '\t', '\n', '\a' are known as Escape Sequence Characters.
- There are several escape sequence characters.
- '\a' (bell), '\b' (backspace), '\n' (new line), '\r' (carrige return),
- '\f' (form feed), '\t' (horizontal tab), '\v' (vertical tab),
- '\''(single quote '), '\"' (double quote "), '\\' (back slash \)
-
- C++ classifies "C++"(narrow character string), L"C++"(wider character string) as strings
-
- C++ classifies 8, 5, -7, 0b101, 0x101, 0101, 101 as int.
-
- C++ classifies 3.14f, 2.71f as float. float is single precision.
- C++ classifies -3.9, 9.8,-4.5 as double. double is double precision.
- C++ classifies true, false as bool.
- Total 7 data types are there in c++.

- **Variables**

```
void main() {
	int a;
	double c, d;
	char ch; bool b; //<- This is not a good coding style
}
```

- In a statement we can define one or more variables.
- when multiple variables are defined in a statement, they will have same data type.

- What are white space characters?
- * Space, tab and new line are called as white space characters.
-
- What are camel case, pascal case and snake case?
- * There are three casing styles prevalent in computing industry viz. camel case, pascal case and snake case.
- * Ex of camel case: memoryFile, toolBar, tabManager
- * Ex of pascal case: PushButton, TextArea, CheckBox
- * Ex of snake case: client_dc, paint_dc, internal_exception

```
void main() {
	int a = 38;
}
```

- What is initialization?
  * Giving initial value to a variable at its definition is called as initialization.

- Can variable come into existence without initial value?
  * Yes. But it will come into existence with garbage value.

```
void main() {
	int a;
	a = 101; //Assignment
}
```

- What is assignment?
  -Giving value to a variable after its definition is called Assignment.

```
void main() {
	auto v = 5;
}
```

- auto defined variable needs initializer

- **Type Qualifiers**

```
void main() {
	signed int a = 123;
	unsigned int b = 123U;
	long int c = 123L;
	short int d = 123;
	unsigned long int e = 123UL;
	unsigned short int f = 123;
	signed long int g = 123;
	signed short int h = 123;

}

/*
* By default int is signed.
* signed and unsigned are mutually exclusive
hence we can use only one of them at a time.
* Similarly, long and short are also mutually
exclusive hence we can use only one of them at a
time.
* When signed, unsigned, long, short is used,
then mentioning of int is optional.
  If int is not mentioned, variable is assumed
to be int.
* All type modifiers (signed, unsigned, long,
short) can be applied to int.
* Size of int and long int is 4 bytes. So there
is no difference between int and long int in
case of MSVC++.
* Size of short int is 2 bytes.
*/


unsigned char b = 123U;
	void main() {
	signed char a = 123;
	long char c = 123L;
	short char d = 123;
	unsigned long char e = 123UL;
	unsigned short char f = 123;
	signed long char g = 123;
	signed short char h = 123;

}

/*
* Type modifiers such as signed and unsigned can
only be applied to char.
	*/`


void main() {
	typedef char tinyint;
	tinyint radius = 65;
	radius = radius * 2;
}

/*
-New types can be constructed out of existing
types using typedef.
-General syntax of typedef is:
 typedef existing-typename new-typename;
	*/


void main() {
	//signed float a = 123;
	//unsigned float b = 123U;
	long float c = 123L;
	//short float d = 123;
	//unsigned long float e = 123UL;
	//unsigned short float f = 123;
	//signed long float g = 123;
	//signed short float h = 123;

}

/*
-only long can be applied to float
*/


void main() {
	//signed double a = 123;
	//unsigned double b = 123U;
	long double c = 123L;
	//short double d = 123;
	//unsigned long double e = 123UL;
	//unsigned short double f = 123;
	//signed long double g = 123;
	//signed short double h = 123;

}

/*
* Only long can be applied to double.
* Size of double and long double is 8 bytes. So
no difference between them.
	*/


void main() {
	//signed wchar a = 123;
	//unsigned wchar b = 123U;
	//long wchar c = 123L;
	//short wchar d = 123;
	//unsigned long wchar e = 123UL;
	//unsigned short wchar f = 123;
	//signed long wchar g = 123;
	//signed short wchar h = 123;

}

/*
* No type qualifier can be applied to wchar_t.
* Size of wchar_t is 2 bytes.
	*/


void main() {
	//signed bool a = 123;
	//unsigned bool b = 123U;
	//long bool c = 123L;
```

```
	//short bool d = 123;
	//unsigned long bool e = 123UL;
	//unsigned short bool f = 123;
	//signed long bool g = 123;
	//signed short bool h = 123;

}

/*
* No type qualifier can be applied to bool.
* Size of bool is 2 bytes.
	*/


void main() {
	const int u = 5;
	int v = u;
	u = 10;
}

/*
- const modifer makes variable a read only
variable.
- A const variable can be initialized but cannot
be assigned.
- Initialization is mandatory.
	*/
```

➢ DAY 02

➢ 01 Input Output

```
Difference between C and C++ Input Output

1. In C, we need to include stdio.h
   In C++, we need to include iostream

2. scanf_s and printf are functions
   cin and cout are objects.

3. For scanf_s, printf respective format
specifiers need to be mentioned.
   No such need in case of cin and cout.

4. Address of the variable need to be
passed to scan_f.
   No such need in case of cin.
```

```
#include<stdio.h>

void main() {
	int a = 0;
	printf("Enter an integer: ");
	scanf_s("%d", &a);

	float r = 0.0f;
	printf("Enter a decimal number: ");
	scanf_s("%f", &r);

	double d = 0.0;
	printf("Enter yet another decimal
number:");
	scanf_s("%lf", &d);

	char c = 0;
	printf("Enter a character: ");
	fflush(stdin);
	scanf_s(" %c", &c, 1);

	wchar_t wc = 0;
	printf("Enter yet another character: ");
	fflush(stdin);
	scanf_s(" %C", &wc, 1);

	printf("value of a is %d\n", a);
	printf("value of r is %f\n", r);
	printf("value of d is %lf\n", d);
	printf("value of c is %c\n", c);
	printf("value of wcis %C\n", wc);
	}
```

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

void main() {
	int a = 0;
	cout << "Enter an integer: ";
	cin >> a;

	float r = 0.0f;
	cout << "Enter a decimal number: ";
	cin >> r;

	double d = 0.0;
	cout << "Enter yet another decimal
number: ";
	cin >> d;


	char c = 0;
	cout << "Enter a character: ";
	cin >> c;

	wchar_t wc = 0;
	wcout << L"Enter yet another character:
";

	wcin >> wc;


	cout << "a= " << a << endl;
	cout << "r= " << r << endl;
	cout << "d= " << d << endl;
	cout << "c= " << c << endl;
	wcout << "w= " << wc << endl;

}

/*
What are cin and cout?
* They are objects.
* Used for input and output.

What is <<?
```

* << is an operator.
* In cout context, << operator is known as insertion or put to operator.

What is >>?
* >> is an operator.
* In cin context, >> operator is known as extraction or get from operator.

What does endl do?
* It's a function.
* It add a new line and flush the output buffer to output device.

Which other function can be located in place of endl?
* flush.

What does flush do?
* It's a function.
* It flushes the output buffer to the output device.
* Note it doesn't add a new line.

02 Operators

Arithmetic Operators

```
void main() {
	int a = 7, b = 3, c = 0;
	c = a + b;
	c = a - b;
	c = a * b;
	c = a / b;
	c = a % b;

	c = 7 / 3; // 2
	c = 7 % 3; // 1
	c = -7 % 3;
	c = 7 % -3;

}

/*
What are arithmetic operators in C++?
- + (addition), - (subtraction), * (multiplication), / (division), % (modulus or remainder) are arithmetic operators.
- They are binary operators. It means they require two operands to operate.
- Modulus operator cannot be used with float or double.
*/
```

Increment and Decrement Operators

```
// Pre increment
void main() {
	int a = 1, b = 0;
	b = ++a;
	}

// Post increment
void main() {
	int a = 1, b = 0;
	b = a++;
	}

// Pre decrement
void main() {
	int  a = 3, b = 0;
	b = --a;

	}

// Post Decrement
void main() {
	int a = 3, b = 0;
	b = a--;
	}
```

Compound Assignment Operators

```
void main() {
    int a = 7, b = 2;
    a += b; // a = a + (b)
    a -= b; // a = a - (b)
    a = b; // a = a (b)
    a /= b; // a = a / (b)
    a %= b; // a = a % (b)

    int c = 3;
    c *= a + b; // c = c*(a + b)
}
```

Relational Operators

```
void main() {
    bool b = false;
    int u = 3, v = 4;
    b = u < v; // '<' less than operator
    b = u > v; // '>' greater than operator
    b = u <= v; // '<=' less than equal to operator
    b = u >= v; // '>=' greater than equal to operator
    b = u != v; // '!=' not equal to operator
    b = u == v; // '==' equal to operator
}
```

Logical Operator

```
void main() {
    bool b = false;
    b = !false;
    b = !true;
    b = !(3 > 4);

    int u = 3;
    b = !(u < 4);

    int v = 4;
    b = !(v > u);
}
```

```
/*
int i = 5; i = !i; i = !i; will result into
1 or 5?
* No. It will set i to 1.
*/

----------------------

void main() {
       bool b = true;
       b = false && false; // result is false
       b = false && true;   // result is false
       b = true && false;   // result is false
       b = true && true;    // result is true

       int c = 3, d = 4, e = 0;
       b = c > e && c < d;
}

/*
How do we specify logical AND operator in C++?
* Using &&.

Is logical AND (&&) operator of C++ smart
operator?
* The logical AND operator of C++ is a smart
operator.
* If lhs expression evaluates to false then rhs
expression is not evaluated at all.
*/

/*
Questions:
Does 0 < a < 10 syntax makes sense in C++?
* No.
   */

------------------------

void main() {
       bool a = true;
       a = false || false;
       a = false || true;
       a = true || false;
       a = true || true;

       int c = 3, d = 4, e = 0;
       a = c > e || c < d;
}

/*
How do we specify logical OR operator in C++?
* Using ||.

Is logical OR (||) operator of C++ smart
operator?
* The logical OR operator of C++ is a smart
operator.
* If lhs expression evaluates to true then rhs
expression is not evaluated at all.
   */
```

- **Comma Operator**

```
void main()
{
       int i = 0, j = 0, k = 0;
       i = 1, j = 5, k = i + j;
}
```

- **Conditional Operator**

```
   void main() {
       int a = 10, b = -32, c = 0;
       c = a < b ? a : b;
   }

   /*
   What is general syntax of "?:" ?
   *variable = condtion ? true - expression :
   false - expression;
   *Condition can be simple or complex.
   * If condition evaluates to true, true -
   expression is evaluated and result is
   assigned to variable.
   * Else false - expression is evaluated and
   result is assigned to variable.
   * Mentioning of variable is optional.
   * /
```

- **Nested Assignment**

```
void main() {
       int a = 0, b = 0, c = 0;
       a = b = c = 1; // Nested assignment as (a
= (b= (c =1)));
          }
```

- **03 Expression Evaluation:**

```
void main() {
       int result = 0;
       result = 3 - 2 * 4; // (result = (3 -
(2*4)));  -5
       result = 3 * 2 / 4; // (result = ((3*2)
/4)); 1

}

/*
- In the first expression since the operators
(=, -, *) belongs to different groups
  it is the precedence of the operators matter.
- In the second expression since the operators
(* and /) belongs to the same group
  it is the associativity of the operators
matter.
- Thus, when operators belong to same group
associativity is considered.
  And if they belong to different groups then
precedence between them is considered.
```

```
*/
```

- **04 Type conversion**

```
void main() {
	int u = 5;
}

/*
* No type conversion involved here
  since at both the ends (variable end and constant end) the data type is same.
	*/
```

```
void main() {
	double u = 5;
}

/*
* The type of 5 is int and the type of u is double, hence type conversion is involved in the above progam.
* It's a implicit kind of type conversion since there is no loss of data while converting from int to double.
* Following convrsions happen automatically
* bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double
*/
```

```
void main() {
	int u = 5.0;
}

/*
* The type of 5.0 is double and the type of u is int.
* Since dissimilar types are involved in the expression, type coversion is involved.
* This type conversion may result into data loss, hence compiler won't do it quietly.
* If conversion involves basic type to basic type then compiler throws warning.
* Explicit casting is essential in such case to get rid of the warning.
* Donot ignore warnings.
*/
```

```
void main() {
	int u = static_cast<int>(5.0);

}

/*
* static_cast is known as casting operator.
* It is used in explicit type conversion.
* By doing explicit type cast, we take responsibility of loss of data due to type conversion to ourselves.
* In such case, compiler stops producing warnings related to the type conversion.
*/
```

- **05 determining type of an expression**

```
void main() {
	int u = 3 + 5; // both expression type and receving variable type is int, no type conversion is involved

	double v = 3 + 5; // expression type is int and receiving variable type is double
	// type conversion is involved. Since type conversion involved is promoting type
	// conversion happens implicitly.

	int w = static_cast<int>(3 + 5.0); // expression type is double and receiving variable type is int.
	// type conversion is involved. Since type conversion involved is demoting type
	// compiler will generate warning. Warning can be supressed using explicit type cast.
}
```

DAY 03

- **01 Functions**
  - **01 Introduction to functions**

```
#include<iostream>

using namespace std;

void main() {

	int a = 0;
	cout << "Enter first number = ";
	cin >> a;

	int b = 0;
	cout << "Enter the second number= ";
	cin >> b;
```

```
        int c = 0;
        c = a + b;
        cout << "The sum of a & b is " << c <<
endl;
        }
```

```
#include<iostream>

using namespace std;

void add() {
        int a = 0;
        cout << "Enter first number = ";
        cin >> a;

        int b = 0;
        cout << "Enter the second number= ";
        cin >> b;

        int c = 0;
        c = a + b;
        cout << "The sum of a & b is " << c
<<endl;

}

void main() {
        add();
}

/*
* Both 'main' and 'Add' are known as functions.
* A function is a named reusable code block.
* 'main' is called caller and 'Add' is called
called.
*/
```

- **02 Passing arguments to functions**

```
#include<iostream>

using namespace std;

void add(int u, int v) {

    int w = 0;
    w = u+v;
    cout << "The sum of a & b is " << w <<
endl;

}

void main() {
    int a = 0;
    cout << "Enter first number = ";
    cin >> a;

    int b = 0;
    cout << "Enter the second number= ";
    cin >> b;

    add(a,b);
}
```

- **03 returning value from function**

```
#include<iostream>

using namespace std;

int add(int u, int v) {

        int w = 0;
        w = u + v;
        return w;


}

void main() {
        int a = 0;
        cout << "Enter first number = ";
        cin >> a;

        int b = 0;
        cout << "Enter the second number= ";
        cin >> b;

        int c = add(a, b);

        cout << "The sum of a & b is " << c <<
endl;
}
```

- **04 Prototypes mandatory in functions**

```
#include<iostream>

using namespace std;

int add(int u, int v); // Functions
Declaration/ Function Prototypes

void main() {
    int a = 0;
    cout << "Enter first number = ";
    cin >> a;

    int b = 0;
    cout << "Enter the second number= ";
    cin >> b;

    int c = add(a, b);                //
Function Call

    cout << "The sum of a & b is " << c <<
endl;
}
```

```
int add(int u, int v) {      //Function
Defination

    int w = 0;
    w = u + v;
    return w;
}

/*
-Function definition can replace function
declaration but vice versa is not true.
-If function definition is to replace
function declaration,
then order of appearance of function
definition and function call matters to
compiler.
Function declaration should appear before
function call.
-Compiler needs Declaration. Compiler
doesn't need definition.
-Linker needs definition.
*/
```

- 05 Returning exit code from main function

```
int main() {
    return 101;
}

/*
* A main function can return a value.
* Return type of main can be void or int.
* Value returned from main is received by
operation system.
* This value is called as exit code.
* Common convention is that if application
ends with exit code 0 it means application
was terminated normally.
* If it ends with exit code other than 0
then one has to refer documentation of
application.
* On console, can be used to retrieve the
exit code reported by the application.
* Writing return statement in main is
optional. In such case, main returns 0 as an
exit code.
*/
```

02 Macros

- 01 Macro Variable

```
#define PI 3.1428 //PI is known as macro
variable

void main() {
       int radius = 10;
       double area = PI * radius * radius;
       double volume = 4.0 / 3.0 * radius *
radius * radius;
}

/*
* When program is compiled, preprocessor
replaces PI by its corresponding value.
* Note macro variable is not a memory variable
hence no memory is reserved for macro variable.
* Macro variable improves program readability
and maintenance.
              */
```

- 02 Macro Functions

```
#define MIN(U,V) (U<V ? U:V) //MIN is a macro
function

void main() {
       int i = 1, j = 2, k = 0;
       k = MIN(i, j); // processor changes k=
MIN(i,j) to k=(i <j ? i: j);
}

/*
* When program is compiled, preprocessor
replaces macro function with its corresponding
code.
* Many leading frameworks use macro functions
for code substitution. For ex. ATL, MFC, CAA
etc.
*/

-----------------------------------------------------------------
---------

#define SWAP(U,V) U = U + V; \
V = U - V; \
U = U - V;

int main() {
       int i = 1, j = 2;
       SWAP(i, j);
}
```

03 Arrays

- 01 array definition
  - **Single dimension**

```
void main() {
       int a[3];
}

/*
* General syntax for defining an array: datatype
array-name[dimension][dimension][dimension]...
* Array occupies continuous memory block.
* Size of an array is equal to size of data type
x number of elements.
```

```
* The array 'a' is a fixed length array. It
means dimension of 'a' cannot be changed at
runtime.
* The dimension has to be mentioned at compile
time.
*/

-----------------------------------------------------------------

----------------------

int main() {
	const int n = 3;
	int a[n];
}


/*
- A non-constant variable cannot be used to
specify array dimension.
*/

int main() {
	const int n = 3;
	int a[n]; //as good as int a[3]
}


/*
* A constant variable can be used to specify
array dimension.
* Most compilers optimize above code.
* They replace constant variable with its value.
* Hence upon optimization, int a[n] is converted
to int a[3].
*/
```

```
# define N 3

void main() {
	int a[N];
}
```

- **02 Double Dimension**

```
#define R 2
#define C 3

void main() {
	int u[2][3];

	const int r = 2;
	const int c = 3;
	int v[r][c];

	int w[R][C];
}
```

- **02 array initialization**

```
#define N 3
#define R 2
#define C 3

int main() {
	int a[N] = { 1,2,3 };
	int b[R][C] = { {1,2,3},{4,5,6} };
	int c[R][C] = { 1,2,3,4,5,6 };

}
```

```
#define N 3
#define R 2
#define C 3

int main() {
	int a[N] = { 1,2 };
	int b[R][C] = { {1,5},{4,5,6} };
	int c[R][C] = { 1,2,5,6,7 };

}
/*
* If number of initializers are less than number
of elements then it doesn't result into
compilation error.
* Elements for which initializers are missing
are set to zero.
*/

---------------------------------------------

#define N 3
#define R 2
#define C 3

int main() {
	int a[N] = { 0 };
	int b[R][C] = { 0 };
	int c[R][C] = { 0 };

}

/*
All elements will be set to zero.
*/


------------------------------------------------
-

#define N 3
#define R 2
#define C 3

int main() {
	int a[N] = { 1 };
	int b[R][C] = { 1 };
	int c[R][C] = { 1 };

}

/*
-This syntax doesn't set all elements to one.
- Only first element is set to one, rest all
elements are set to zero.
*/
```

```
---------------------------------------------------
---

#define C 3

int main() {
	int a[] = { 1,2,3,4 };
	int b[][C] = { {1,2,3},{4,5,6,} };
	int c[][C] = { 1,2,3,4,5,6,7 };

}

/*
* C++ can use initializers to fix dimension of
an array.
* For double dimension array, column dimension
is mandatory.
* Compiler can figure out row dimension based
upon the number of initializers.
*/
```

- **03 accessing the elements**

```
#define N 3

int main() {
	int a[N] = { 0 };
	a[0] = 1;
	a[1] = 2;
	a[2] = a[0] + a[1];
}
```

- **04 selective programming**
  - **01 if**

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

void main() {
	int n = 0;
	cout << "Input a Number: ";
	cin >> n;

	int result = n;
	if (n < 0)
		result = -n;

	cout << "Absolute value of " << n << " is
" << result << endl;
}

/*
- Simple if statement.
*/

------------------------------------------------

#include <iostream>
using namespace std;

void main() {
	int x1 = 0;
	cout << "Input x1: ";
	cin >> x1;

	int y1 = 0;
	cout << "Input y1: ";
	cin >> y1;

	int x2 = 0;
	cout << "Input x2: ";
	cin >> x2;

	int y2 = 0;
	cout << "Input y2: ";
	cin >> y2;

	if (x2 < x1) {
		int t = x1;
		x1 = x2;
		x2 = t;
	}

	if (y2 < y1) {
		int t = y1;
		y1 = y2;
		y2 = t;
	}
}

/*
- If with a block of statements
*/
---------------------------------------------------------------

#include <iostream>
using namespace std;

void main() {
	int a = 0;
	cout << "Input coefficient a: ";
	cin >> a;

	int b = 0;
	cout << "Input coefficient b: ";
	cin >> b;

	int c = 0;
	cout << "Input coefficient c: ";
	cin >> c;

	int discriminant = b * b - 4 * a * c;

	double root1 = 0.0, root2 = 0.0;
	if (discriminant >= 0) {
		root1 = (-b - sqrt(discriminant))
/ (2 * a);
		root2 = (-b + sqrt(discriminant))
/ (2 * a);
	}
	else {
		cout << "Real roots doesn't exist"
<< endl;
	}
}
```

```
/*
- If else statement.
*/
```

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

void main() {
	int a = 0;
	cout << "Enter first integer: ";
	cin >> a;

	int b = 0;
	cout << "Enter second integer: ";
	cin >> b;

	int c = 0;
	cout << "Enter third integer: ";
	cin >> c;

	int largest = 0;

	if (a > b)
		if (a > c)
			largest = a; // a = 5, b = 3, c = 2
		else
			largest = c; // a = 3, b = 2, c = 5
	else
		if (b > c)
			largest = b; // a = 2, b = 5, c = 3
		else
			largest = c; // a = 2, b = 3, c = 5

	cout << "Largest value between " << a << ", " << b << " and " << c << " is " << largest << endl;
}

/*
* Nesting if...else statements.
*/
```

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

void main() {
	int a = 0;
	cout << "Enter first integer: ";
	cin >> a;

	int b = 0;
	cout << "Enter second integer: ";
	cin >> b;

	int c = 0;
	cout << "Enter third integer: ";
	cin >> c;

	int largest = 0;

	if (a > b && a > c)
		largest = a; // a = 5, b = 3, c = 2
	else if (b > c)
		largest = b; // a = 2, b = 5, c = 3
	else
		largest = c; // a = 2, b = 3, c = 5

	cout << "Largest value between " << a << ", " << b << " and " << c << " is " << largest << endl;
}

/*
- if...else if ladder
- multiway conditional statement
*/
```

- **02 Switch**

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

int main() {
	double a = 0;
	cout << "Enter first number: ";
	cin >> a;

	double b = 0;
	cout << "Enter second number: ";
	cin >> b;

	char op;
	cout << "Enter operator [+ - * /]: ";
	cin >> op;

	double result = 0.0;

	switch (op) {
	case '+':
		result = a + b;
		cout << "result = " << result << endl;
		break;
	case '-':
		result = a - b;
		cout << "result = " << result << endl;
		break;
	case '*':
	case 'x':
	case 'X':
		result = a * b;
		cout << "result = " << result << endl;
		break;
	case '/':
```

```
            result = static_cast<double>(a) /
b;
            cout << "result = " << result <<
endl;
            break;
        default:
            cout << "Invalid operator." <<
endl;
        }
}

/*
- Switch variable can be int, char, wchar_t,
bool or enum type.
  It cannot be float or double typed.
- A fall through happens when case is not
terminated with break or return.
- default can be placed anywhere within switch
... case block.
  If placed at the beginning or in between then
it has to be terminated with break or return.
- Good practice is to place default block at the
end of switch case block.
  In that case no break or return statement is
required.
- If no case matches with switch variable then
default block is executed.
  If matching case exist then default is not
executed irrespective of its position.
- Note default is optional.
- Only one matching case or default is executed.
After which control exits the switch...case.
- In some instances, cases are likely to be
terminated with return statement.
        */
        -----------------------------------------
        ---------------------

int main() {
        double d = 1.0;
        switch (d) {
        case 1.0:
            break;
        case 2.0:
            break;

        }
}



        //double and float cannot be used for
        switch in c++.
```

- **05 repetitive programming**
    - **01 while**

-> Infinite loop

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

void main() {
        while (true)
            cout << "Hello, World" << endl;
}

/*
* while is a pre test loop.
* Pre test loop is a loop in which test
expression is evaluated first and then the loop
body is executed.
* If test expression evaluates to true, the loop
body is repeated else it is terminated.
* while(true) results into an infinite loop.
*/
```

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

void main() {
        while (1)
            cout << "Hello, World" << endl;
}

/*
* while(1) results into an infinite loop.
*/
```

-> finite loop

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

void main() {
        int i = 0;
        while (i < 3) {
            cout << "Hello, World" << endl;
            i++;
        }
}

/*
* Finite loop.
* while loop terminates, when test expression
evaluates to false, .
* Typically, while loop is used in a situation,
  when number of iterations are not known in
advance.
*/
```

- **02 do while**

-> Infinite loop

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

void main() {
        do
            cout << "Hello, World" << endl;
        while (true);
}

/*
* do...while is a post test loop.
```

```
* Post test loop is a loop in which test
expression is evaluated after the completion of
the loop body.
* If test expression evaluates to true, the loop
body is repeated else it is terminated.
* do ... while(true) results into an infinite
loop.
*/
```

-> Finite loop

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

void main() {
	int i = 0;
	do {
		cout << "Hello, World" << endl;
		i++;
	} while (i < 3);
}

/*
* do...while loop is a post test loop.
* First iteration is sure. Next iteration
depends upon the result of test expression.
* If test expression evaluates to true then
do...while loop body is repeated else it is
terminated.
*/
```

- **03 for loop**

-> Infinite loop

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

void main() {
	for(;;)
		cout << "Hello, World" << endl;
}

/*
* for is a pre test loop.
* for(;;) results into an infinite loop.
*/
```

-> Finite loop

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

void main() {
	int i = 0;
	for (i = 0; i < 3; ++i) {
		cout << "Hello, World" << endl;
	}
	cout << "i = " << i << endl;
}

/*
What is general systax of for?
* for(initialization; cond; expression) {
	statement-1;
	statement-2;
	statement-3;
	}

* All sections are optional but ';' separating
sections are not optional.
* If loop variable is defined outside the loop
then it remains in the scope after the
termination of the for loop.
* for is to be used when number of iterations to
be perfomed are known in advance.
*/
```

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

void main() {
	for (int i = 0; i < 3; ++i) {
		cout << "Hello, World" << endl;
	}
	//cout << "i = " << i << endl;
}

/*
* If loop variable is defined inside for loop
then its scope gets over as soon as loop is
terminated.
*/
```

DAY 05

- **01 Function overloading**

```
int add(int n1, int n2);
double add(double n1, double n2);

int main() {
	int a = 1, b = 2, c = 0;
	c = add(a, b);

	double i = 1.0, j = 2.0, k = 0.0;
	k = add(i, j);

}

int add(int n1, int n2) {
	return n1 + n2;
}
double add(double n1, double n2) {
	return n1 + n2;
}


/*
* Function overloading is a feature of c++.
* When two or more functions share same name
but differ in
```

```
  parameter list, then those functions are
said to form function overloading.
* The difference in the parameter list must
be in the terms of
  type difference of parameters or number of
parameters or
  the order of their types and not in terms
of their names.
* When C++ compiler compiles CPP code, it
modifies function name.
  This phenomenon is known as name
mangling(or name decoration). While mangling
name,
  it considers following aspects of the
function:
  1. Name of the function
  2. Number of Parameters
  3. Types of Parameters
  4. Order of Parameters Types
  5. Namespace
  6. const clause and access speicifer in
case member functions etc.
  C++ uses name mangling to provide function
overloading feature.
*/
```

```
int f() { return 1; }

double f() { return 1.0; }

int main() {
	int a = f();
	double b = f();

}

/*
* Function overloading is impossible between
two functions
  that just have a different return type.
*/
/*
* Function overloading cannot occur between
two functions
  that only differ in return type.
	*/
```

- **02 Inline function**

add.cpp

```
#include "add.h"
```

Add.h

```
#pragma once

 inline int add(int u, int v);

 int add(int u, int v) {
	return u + v;
 }
```

Source01.cpp

```
int add(int u, int v);

int main() {
	int result = 0;
	result = 3 + 5; //Expression is inline
	result = add(3, 5); //add is not
inline
}

int add(int u, int v) {
	return u + v;
}
```

Source02.cpp

```
inline int add(int u, int v);

int main() {
	int result = 0;
	result = 3 + 5; //Expression is inline
	result = add(3, 5); //add is not
inline
}

inline int add(int u, int v) {
	return u + v;
}

/*
- 'inline' is a C++ feature.
- C++ compiler replaces call by the
definition of the inline function.
- Note inline is a request to the compiler.
- If compiler observes its not feasible to
inline the definition then it keeps
  call to function as it is.
- If function is lengthy, virtual or recusive
then compiler doesn't do inlining.
*/
```

Source 03.cpp

```
#include "add.h"
int main() {
	int result = 0;
	result = 3 + 5; //Expression is inline
	result = add(3, 5); //add is not
inline
}

//for inline functions, function declarartion
and defination should be in header files.
```

03 Inline function vs macro function

```
#define MIN(i,j)(i<j?i:j)

inline int min(int i, int j) {
	return (i < j ? i : j);
}

int main() {
	int a = 1, b = 2, c = 0;
	c = MIN(++a, ++b); //a=2, b=3, c=2

	int u = 1, v = 2, w = 0;
	w = min(++u, ++v); //a=2, b=3, c=2
}

/*
- Macro expansion is done by preprocessor.
- Inline function expansion is done by compilation phase.
- If expression such as ++a is passed to macro, then that expression
  is substituted as it is whereas in case of inline functio,
  the result of the expression is substituted and not the expression
  itself.
- The parameters of macro function doesn't have data types.
  Whereas inline function parameters have data type specification.
  So inline functions are more type safe then macros.
	*/
```

04 default arguments

```
int add(int u, int v) {
	return u + v;
}

int main() {
	int result = 0;
	result = add(1, 2);
	result = add(5);
}

/*
- After parsing, above code, compiler realizes that
  add is a function and it takes two parameters of int type.
  Both parameters are mandatory parameters. Hence arguments to them are compulsory.
  In the 2nd call to add, second argument is missing hence compiler throws
  too few arguments error on that line.
	*/

	--------------------------

int add(int u, int v=0) {
	return u + v;
}

int main() {
	int result = 0;
	result = add(1, 2);
	result = add(5);
}

/*
-C++ has a feature called default argument using which default value can be given
to the parameter(as given to 'v' of 'add' in the above program).
- Assigning default value to the parameter makes that parameter optional.
- If specific argument is mentioned to that parameter in the call,
then that value is assigned to that paramter.
- If no argument is mentioned to that parameter in the call,
then default value is assinged to that parameter.
	* /

	------------------------------

int add(int u, int v)

int main() {
	int result = 0;
	result = add(1, 2);
	result = add(5);
}

int add(int u, int v = 0) {
	return u + v;
}

/*
- Though the default argument is mentioned in the definition
  still we see compiler is giving error.
- This is because, compiler compiles code top to bottom.
- Compiler learns from the prototype that both 'u' and 'v'
  are mandatory parameters, so when it sees call to add
  with single argument it realizes the break in the add function protocol
  hence it throws error.
	*/

	------------------------------------

int add(int u, int v=0)

int main() {
	int result = 0;
	result = add(1, 2);
```

```
        result = add(5);
}

int add(int u, int v = 0) {
        return u + v;
}

/*
- Default argument cannot be mentioned in
function decalaration and
  function definition at the same time.
*/
```

```
int add(int u, int v = 0);

int main() {
        int result = 0;
        result = add(1, 2);
        result = add(5);
}

int add(int u, int v) {
        return u + v;
}

/*
A default argument is a value provided in
function declaration
that is automatically assigned by the
compiler to the parameter
if in case caller doesn't provide an argument
for the respective
parameter.

A default argument makes parameter as
optional parameter.When
default argument is not mentioned, the
parameter is mandatory
parameter.A mandatory parameter must be
assigned an argument in the
function call.

A default argument must be set in the
function prototype / declaration.

A redifinition error would happen if default
argument is assigned
both in function declaractionand definition.
        * /
```

```
int add(int u=0, int v);

int main() {
        int result = 0;
        result = add(1, 2);
        result = add(5);
}

int add(int u, int v) {
        return u + v;
}
/*
A function can have one or more optional
parameters.
They should all appear after the mandatory
parameters.

In case we wish to provide specific value to
an optional parameter
which is far in the parameter list, then it
is compulsory to give
arguments (though default) to all optional
parameters in between.
        */
```

- **05 recursive function**

```
long factorial(int n);

int main() {
        int result = 0;
        result = factorial(3);
}

long factorial(int n) {
        if (n == 1)
                return 1;
        return n * factorial(n - 1);
}

/*
- A function calling itself is called a
recursive function.
- Too deep recursion may result into stack
overflow error.
        */
```

DAY 06

- **01 Storage classes**
  - **01 non static local (auto) variable**

```
void f() {
    int i = 1;
}

int main() {
    f();
    i = 5; // attempting to access "i" of
"f" which is not possible
}
/*
- A local variable is a variable defined
within the block.
- 'i' is a local variable of function 'f'.
- 'i' is a non-static local variable.
```

```
    - Non-static local variable are also
    called as automatic variables.
    - The scope of local variable is limited
    to the block in which it is defined.
    - 'i' is a local variable of 'f' hence its
    scope is limited to the 'f'.
    - It cannot be accessed in any other
    function.
    */

    -------------------------------
void f() {
	int i = 1;
}

int main() {
	int i = 5;
	f();
}
/*
- Can two functions have local variables with
same name?
- Yes. Both 'f' and 'main' can have their own
local varibale 'i'.
- Both 'i's are treated as different.
- Non-static local variables are located in
stack frame.
*/

    -----------------------------

#include <iostream>
using namespace std;

void f()
{
	int i = 1;
	cout << i << endl;
	i = i + 1;
}

int main() {
	f();
	f();
	f();
}

/*
- Lifetime of local variable is limited to
the block.
- Everytime control exits 'f', 'i' variable
is destroyed and
  is created freshly when control enters 'f'
again.
- Since 'i' is getting created freshly, it is
initialized to 1.
- Hence we get output 1 1 1 when above
program is executed.
    */
```

- **02 non static global variables**

```
int i = 1;

void f() {
	i = 5;
}

int main() {
	i = 10;
	f();
}

/*
- A global variable is a variable defined
outside all functions.
- 'i' is a global variable.
- The scope and lifetime of the global
variable is entire application.
- Global variables cease to exist as soon as
application execution terminates.
- An application may be composed of multiple
source files.
- Global variable are accessible across all
source files.
- Gloabl variables are located in static
space (which is also known as data segment)
  or global data space.
*/
```

- **03 External variable**

```
int i = 1;
void k();

void f() {
    i = 5;
}

int main() {
    i = 10;
    f();
    k();
}
/*
- While working with global variable, we
define global variable
  in one implementation file (.cpp) and
declare in header file.
  And we include that header in other
implementation files as needed.
- The declaration of a global variable is
done with the help of 'extern' keyword.
- Note only global variable can be
declared i.e. local variable cannot be
declared,
  it can only be defined.
- What is the difference between
declaration and defintion?
  When a symbol is defined, memory
resource is consumed.
  When a symbol is declared, no memory
resource is consumed.
*/

-------------------------------

extern int i;

void k() {
```

```
    i = 15;
}
```

- 04 static local variable

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

void f()
{
    static int i = 1;
    cout << i << endl;
    i = i + 1;
}

int main() {
    f();
    f();
    f();
}
/*
- Variable 'i' of 'f' is called as static
local variable.
- The lifetime of static local variable is
application wide.
- The scope however is limited to the
block in which it is defined.
- Static local variables are located in
static space or global data space.
- Note any variable located in
static/global space is brought into
  existance as soon as application begins
execution. It then remains
  in memory till the end of an
application.
- Also note, any variable having lifetime
application wide doesn't
  mean is accessible to the entire
application.
*/
```

- 05 Static global variable

```
static int i = 1;
void k();

void f() {
    i = 5;
}

int main() {
    i = 10;
    f();
    k();
}

/*
- The lifetime of a static global variable
is application wide.
- The scope however is translation unit
wide in which it is defined.
- The location is global/static space.
    - If multiple source files define static
    global variable with same
      name then it is perfectly valid. In that
    case each translation unit
      will have its own copy of respective
    global variable.
    - Preprocessed source file is called as
    translation unit.
    */

    --------------------------

int i;

void k() {
        i = 15;
    }
```

- 02 Scope resolution operator

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

int a = 5;
int main() {
        int a = 2;
        cout << a << endl;
        cout << ::a << endl;
}

/*
- Between local identifier and global
identifier with same name,
  local dentifier always takes precedence
over global identifier.
- In such case, to access global identifier,
  we use scope resolution operator (::).
        */
```

- 03 Pointers
  - 01 Obtaining address of a variable

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

int main() {
    int a = 1;
    cout << a << endl;
    cout << &a << endl;
}

/*
- 'a' delivers value of a variable.
- '&a' delivers address of a variable.
*/
```

  - 02 The pointer variable

```
void main() {
    int* pa;
}
/*
```

```
- 'pa' is a pointer variable.
- Its purpose is to hold address of a
variable.
- In above program 'pa' is not initialized
with particular address or null.
- Such pointer is called as 'Wild
Pointer'.
*/
```

- **03 Initialization and assignment of pointer**

```
int main() {
    int a = 1;
    int* pa = &a; // pointer initilization
}

--------------------------

int main() {
        int a = 1;
        int* pa = nullptr;  // pointer
initialization
        pa = &a; // pointer assignment
}

/*
- When we do not have any particular address
to store
  in pointer variable then we prefer to store
nullptr(0)
  in pointer variable.
- Such pointer is called as 'null pointer'.
    *
```

- **04 dereferencing pointers**

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

int main() {
    int a = 1;
    int* pa = &a;

    cout << "a= " << a << endl;
    cout << "pa= " << pa << endl;
    cout << "*pa= " << *pa << endl;

    a = 2; //direct access
    cout << "a= " << a << endl;
    cout << "pa= " << pa << endl;
    cout << "*pa= " << *pa << endl;

    *pa = 5; //Indirect access
    cout << "a= " << a << endl;
    cout << "pa= " << pa << endl;
    cout << "*pa= " << *pa << endl;
}
```

- **05 pointer to array**

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

int main() {
    int a[] = { 1,2,3 };
    cout << a << endl;

    int* pa = a; //a pointer can point to
any array
    cout << pa << endl;
}
```

- **06 Pointer to particular element of an array**

```
int main() {
    int a[] = { 1,2,3 };
    int* pa = &a[1];
}
```

- **07 pointer arithmetic**

```
int main() {
    int a[] = { 1, 2, 5 };
    int* pa = &a[1];
    pa = pa + 1;
    pa = pa - 2;

    int* pb = ++pa;
    pb = pa++;
    pb = pa--;
    pb = --pa;

    pa = &a[0];
    pb = &a[2];
    int diff = pb - pa;

    int n = 0;
    pa = &a[1];
    n = ++ * pa;
    n = *++pa;

    pa = &a[0];
    n = *pa++;
    n = (*pa)++;

    pa = &a[1];
    n = -- * pa;
    n = *--pa;

    pa = &a[2];
    n = *pa--;
    n = (*pa)--;

    pa = &a[0];
    pa += 2;
    pa -= 1;
}

/*
The pa + 1 expression increments address
contained in pointer by four because,
pointer is pointing to 'int'. And size of
'int' is four.

Had pointer pointing to double the same
expression would have incremented pointer
by eight.
```

```
*/
```

DAY 07

- **01 passing and returning to and from function**
  - **01 passing arguments by a value to a function**

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


void print(int n);

int main() {
    int u = 1;
    print (u);
}

void print(int n) {
    cout << n << endl;
    n = 5;
}

/*
- 'u' is said to be passed by value to
'n'.
- Note modification done to 'n' in 'Print'
function will not be updated in 'u'.
- Paramter names can be skipped in the
prototype but it is not recommended to do
so.
*/
```

- **02 returning value from a function**

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


void print(int n);

int main() {
    int u = 1;
    print (u);
}

void print(int n) {
    cout << n << endl;
    n = 5;
}

/*
- 'u' is said to be passed by value to
'n'.
- Note modification done to 'n' in 'Print'
function will not be updated in 'u'.
- Paramter names can be skipped in the
prototype but it is not recommended to do
so.
*/
```

- **03 passing arguments by address to a function**

```
void swap(int u, int v);

int main() {
    int a = 1, b = 2;
    swap(a, b);
}

void swap(int  u, int  v) {
    int t = u;
    u = v;
    v = t;
}

/*
- Passing arguments by value to 'swap'
function doesn't result
  in swapping of their values.
*/

------------------------

void swap(int *u, int* v);

int main() {
        int a = 1, b = 2;
        swap(&a, &b);
}

void swap(int * u, int  *v) {
        int t = *u;
        *u = *v;
        *v = t;
}

/*
-Passing arguments by address to 'swap
function' results into swapping of their
values.
*/
```

- **04 returning address from a function**

```
int g = 1;

int* f();

int main() {
    g = 5;
    int* pg = f(); //Before call int* pg
=f(); after call int *pg =t;
    *pg = 10;
}

int* /* t */ f() {
    return &g;          //int* t =&g;
}
```

```
int* f();

int main() {
	//g = 5;
	int* pg = f(); //Before call int* pg =f(); after call int *pg =t;
	*pg = 10;
}

int* /* t */ f() {

	int g = 1;
	return &g;          //int* t =&g;
}

/*
-Do not return address of a local non-static variable.
*/
```

- **02 Pointers**
  - **01 Strings**

```
int main() {
    char s[] = "Hello, World";
    const char* ps = "Hello, World";
    const char* ps2 = "Hello, World";

    s[7] = 'w';
    //ps[7]= 'w'; //results into compilation error
}

/*
- The string literal such as "Hello, World" is stored in constant memory.
- It is terminated by a null character.
- When char s[] = "Hello, World" statment is executed, a copy of
  "Hello, World" is stored in an array. Thus there exist two
  "Hello, World" strings in the process, one in the constant
  memory and other in the array 's' in stack memory.
  The string stored in 's' can be modified.
- When const char * ps = "Hello, World" statement is executed, the pointer
  'ps' is filled with the address of memory location where
  "Hello, World" string literal is located in constant memory.
  Thus no local copy of "Hello, World" is created like 's'.
  The string cannot be modified, since it is stored in constant memory which is 'read only'.
- Why string stored in constant memory is not allowed to modify?
  It's possible to have more than one pointer to point to string stored in constant memory.
  If modification is allowed, they all will see the modification.
  And probably, this is what a developer doesn't want.
  Hence modifications to string literals is not allowed.
  Such string literals are called as immutable strings.
*/
```

  - **02 string processing library function**

Person.cpp

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

int Average(int* u, size_t v);

int main() {

	int a[5] = { 2,4,6,8,10 };

	size_t size = sizeof(a) / sizeof(int);
	cout<< Average(a, size);
}

int sum = 0;
int Average(int* u, size_t v) {

	for (size_t i = 0; i < v; i++) {

		sum = sum + u[i];
	}

	return sum / v;

}
```

```
/* strlen example */
#include <stdio.h>
#include <string.h>

int main()
{
	const size_t size = 256;
	char szInput[256];
	printf("Enter a sentence: ");
	gets_s(szInput);
	printf("The sentence entered is %u characters long.\n", strlen(szInput));
	return 0;
}
```

```
/* strcpy example */
#include <stdio.h>
#include <string.h>
```

```
int main() {
	char str1[] = "Sample string";
	char str2[40];
	char str3[40];
	strcpy_s(str2, 40, str1);
	strcpy_s(str3, 40, "copy successful");
	printf("str1: %s\nstr2: %s\nstr3: 
%s\n", str1, str2, str3);
}
```

```
/* strcat example */
#include <stdio.h>
#include <string.h>

int main() {
	const size_t size = 80;
	char str[size] = { 0 };
	strcpy_s(str, size, "these ");
	strcat_s(str, size, "strings ");
	strcat_s(str, size, "are ");
	strcat_s(str, size, "concatenated.");
	puts(str);
}
```

```
#include <stdio.h>
#include <string.h>

int main() {
	const size_t size = 80;
	char buffer[size] = { 0 };
	char key[] = "apple";
	do {
		printf("Guess my favorite fruit? 
");

		fflush(stdout);
		scanf_s("%79s", buffer, size);
	} while (strcmp(key, buffer) != 0);
	puts("Correct answer!");
}
```

- **03 array of pointers**

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

int main() {
    const char* ps[] = { "Hello, World", 
"Hi, World" };
    size_t size = sizeof(ps) / 
sizeof(const char*);

    /* following loop prints each string*/
    for (size_t i = 0; i < size; i++)
        cout << ps[i] << endl;

    /* followig loop prints first 
character of each string*/
    for (size_t i = 0; i < size; i++)
        cout << *ps[i] << endl;

    /* followig loop prints adress of each 
string*/
    for (size_t i = 0; i < size; i++)
        cout << static_cast<const 
void*>(ps[i]) << endl;

}
/*
* 'ps' is referred as array of pointers.
* There are two pointers in 'ps', since 
there are two string literals as 
initializers.
* Followings are the interpretation of 
various expressions:
* Just 'ps' returns base address of entire 
array.
* 'ps[i]' returns base address of ith 
string.
* '*ps[i]' returns first character of ith 
string
 *There is special treatement given by 
cout to char * or const char *.
* It prints entire string and not the 
content of the pointer which is address.
* To print content of the pointer char  or 
const char  will have to typecasted to 
'void ' or 'const void *'.
*/

--------------------------------------
```

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

void print(const char* pb[], size_t u);
int main() {
	const char* ps[] = { "Hello, World", 
"Hi, World" };
	size_t size = sizeof(ps) / 
sizeof(const char*);
	print(ps, size);

}

void print(const char* pb[], size_t u) {
	for (size_t i = 0; i < u; i++)
		cout << pb[i] << endl;

}
```

- **04 command line arguments**

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

int main(int argc, char* argv[]) {
    for (int i = 0; i < argc; i++)
        cout << argv[i]<<endl ;
}

/*
- Note argc and argv are not keywords.
```

```
    - argc and argv can be replaced by any
    other name.
    - Command line arguments can be inputted
    from Visual Studio.
      Click Project > Properties > Debugging.
      Specify arguments in Command Line
    Arguments property.
    - Note program always receive command line
    arguments as strings
      i.e. an integer, a real number, boolean
    value, date or any other
      value is received as string.
      Hence to process such argument type
    conversion is required from
      string to corresponding type.
    - First argument is always full name of
    the executable.
    */

    ------------------------------

#include<cstdlib>
#include<iostream>

using namespace std;

int main(int argc, char* argv[]) {
	int n = 0, sum = 0;

	for (int i = 1; i < argc; i++) {
		n = atoi(argv[i]);  //atoi is a
function which converts string character into
integers on which can perform arithmetic
operation.
		sum += n;
	}
	cout << sum << endl;
    }
```

05 Generic pointers

```
int main() {
    int i = 1;
    int* pi = &i;

    double d = 3.14;
    double* pd = &d;

    pi = &d;
}

/*
- Assuming 'i' is located at 100, there
are two facts associated with it:
  - 100 is an address.
  - @ 100, int is stored.

  - Why we could assign address of 'i' to
'pi'?
  It is because 'pi' is compatible with
facts of address 100.
- Assuming 'd' is located at 104, there
are two facts associated with it:
  - 104 is an address.
  - @ 104, double is stored.

  - Why we could assign address of 'd' to
'pd'?
  It is because 'pd' is compatible with
facts of address 104.

  - Why we couldn't assign address of 'd'
to 'pi'?
  'pi' is not compatible with facts of
address 104.
*/
```

```
int main() {
	int i = 1;
	int* pi = &i;

	double d = 3.14;
	double* pd = &d;

	pi = &d;
}

/*
- Assuming 'i' is located at 100, there are
two facts associated with it:
  - 100 is an address.
  - @ 100, int is stored.

  - Why we could assign address of 'i' to
'pi'?
  It is because 'pi' is compatible with facts
of address 100.
- Assuming 'd' is located at 104, there are
two facts associated with it:
  - 104 is an address.
  - @ 104, double is stored.

  - Why we could assign address of 'd' to
'pd'?
  It is because 'pd' is compatible with facts
of address 104.

  - Why we couldn't assign address of 'd' to
'pi'?
  'pi' is not compatible with facts of
address 104.
*/
```

06 pointer to function

```
void f();
int g(int n);
void k();
int h(int k);

int main() {
    void(*fptr)() = f;
    f();         //direct call to 'f'
```

```
    fptr();            // indirecdt call
to 'f'

    int (*gptr)(int) = g;
    g(1);        //direct call to 'g';
    gptr(5);     //indirect call to 'g'

    k();         //direct call to 'k'
    fptr = k;
    fptr();

    gptr = h;
    gptr(6);

}

void f() {
}

int g(int n) { return n; }
void k(){}

int h(int k) { return k; }
```

```
int getsequential() {
      static int i = 0;
      i++;
      return i;
}

int getEven() {
      static int i = 0;
      i = i + 2;
      return i;
}

void fill(int arr[], size_t size,
int(*getrange)()) {
      for (size_t i = 0; i < size; i++) {
            arr[i] = getrange();
      }
}


      int main() {

            size_t s = 5;
            int a[5] = { 0 };

            fill(a,s,getEven);

      }
```

- *DAY 07*

- **01 Dynamic memory management using new**

```
int main() {
	int* pn = new int;
	*pn = 1;
	delete pn;
	pn = nullptr;
}

/*
1. malloc is not a keyword, new is a keyword.
2. malloc is a function, new is a operator.
3. malloc needs to be passed number of bytes to reserve, the number of bytes to be reserved is passed automatically to new by compiler.
4. explicit type casting is required on return value of malloc, no such type casting is required on the value returned by new
5. memory reserved using malloc should be freed using free function. memory reserved using new should be freed using delete operator.
6. In case of malloc...free inclusion of malloc.h is reuqired. In case of new delete inclusion of particular header file is not required.
7. malloc...free can be invoked in c/c++ programs. new and delete can be invoked only in c++ programs.
8. malloc doesn't invoke constructor where as new invokes constructor.
   free doesn't invoke destructor where as delete invokes destructor.
   It is because of this in c++ call to malloc...free is discouraged.

'new int' allocates one int space (4 bytes) on free store.
'delete pn' releases same space allocated on free store.
Note 'delete pn' doesn't delete 'pn' itself, it releases
memory pointed by pn.

It's possible that 'new' may fail to allocate space on free store.
In such case, an exception 'std::bad_alloc' is thrown.
If malloc fails it returns null.
*/

int main() {
	int* pn = new int(1);
	delete pn;
	pn = nullptr;
}
/*
- The argument passed after the type in new expression is used
  to initialize memory block allocated by new.
*/
```

```
#include <iostream>

int main() {
	int count = 5;
	int* pn = new int[count];

	for (int i = 0; i < count; i++)
		pn[i] = i;

	for (int i = 0; i < count; i++)
		*(pn + i) = 2 * i;

	delete[] pn;
	pn = nullptr;
}
/*
- Use same form of new and delete.
  If [] is used with new then use [] with delete.
  If [] is not used with new then do not use it with delete.
	*/

------

int main() {
int* pn = nullptr;

delete pn; // it's ok to pass null pointer to delete operator

pn = new int;
delete pn; // it's ok to pass a valid pointer to delete operator
pn = nullptr;

pn = new int;
int* pn2 = pn;
delete pn; // this results into pn2 as dangling pointer
pn = nullptr;

delete pn2; // it's NOT OK to pass dangling pointer to delete operator
}
```

- **02 memory leakage**

```
#include <crtdbg.h>

int main() {
	int* pn = new int;
	*pn = 1;
	//delete pn;
	pn = nullptr;
	_CrtDumpMemoryLeaks();
}
```

```
#include <crtdbg.h>

int main() {
	int* pn = new int;
	*pn = 1;
	pn = nullptr;

	pn = new int;
	*pn = 2;
	delete pn;
	pn = nullptr;

	_CrtDumpMemoryLeaks();
	}
```

```
#include <crtdbg.h>

int* CreateInt(int value);
void Release(int* ptr);

int main() {
	int* ptr = CreateInt(1);
	Release(ptr);
	ptr = CreateInt(5);
	Release(ptr);
	_CrtDumpMemoryLeaks();
}

int* CreateInt(int value) {
	int* pn = new int(value);
	return pn;
}

void Release(int* ptr) {
	delete ptr;
	ptr = nullptr;
}
```

- **03 dangling pointer**

```
int main() {
	int* pn = new int(1);
	int* pn2 = pn;
	delete pn;
	pn = nullptr;
}

/*
- 'pn2' is a dangling or floating pointer.
*/
```

- **04 pointer to pointer**

```
int main() {
	int a = 1;
	int* pa = &a;
	int** ppa = &pa;

	a = 5;
	*pa = 10;
	**ppa = 15;

    }
```

```
void alloc(int* pn) {
	pn = new int;
}

int main() {
	int* pa = nullptr;
	alloc(pa);
	delete pa;
	pa = nullptr;
    }
```

```
void alloc(int** pn) {
	*pn = new int;
}

int main() {
	int* pa = nullptr;
	alloc( &pa);
	delete pa;
	pa = nullptr;
    }
```

➢ **05 pointer and const**

```
int main() {
        const int u = 1;

        int *p = &u;

        u = u + 1;

        *p = *p + 1;

        int v = 2;

        p = &v;
}
```

```
int main() {
        int u = 1;
        // Y

        const int* p = &u;              // Y

        u = u + 1;
        // Y

        *p = *p + 1;                    // Y

        int v = 2;
        // Y

        p = &v;         // Y
        }
```

```
int main() {
        const int u = 1;        // Y

        const int* p = &u;              // Y

        u = u + 1;
        // Y

        *p = *p + 1;                    // Y

        int v = 2;
        // Y

        p = &v;         // Y
        }
```

```
int main() {
        const int u = 1;        // Y

        int const* p = &u;              // Y

        u = u + 1;
        // Y

        *p = *p + 1;                    // Y

        int v = 2;
        // Y

        p = &v; // Y
        }
```

```
int main() {
        int u = 1;
        // Y

        int* const p = &u;              // Y

        u = u + 1;
        // Y

        *p = *p + 1;                    // Y

        int v = 2;
        // Y

        p = &v;         // N
        }
```

```
int main() {
        const int u = 1;        // Y

        int* const p = &u;              // Y

        u = u + 1;
        // Y

        *p = *p + 1;                    // Y

        int v = 2;
        // Y

        p = &v;         // Y
        }
```

```
int main() {
        int u = 1;
        // Y

        const int* const p = &u; // Y

        u = u + 1;              // Y


        *p = *p + 1;      // Y

        int v = 2;              // Y


        p = &v;           // Y

        }
```

```
int main() {
        const int u = 1;
        // Y

        const int* const p = &u;           // Y

        u = u + 1;              // N
        *p = *p + 1;      // N

        int v = 2;        // Y


        p = &v; // N

        }
```

- **06 Reference to scalar**

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

int main() {
    int i = 1;
    int j = i;
    i++;
    j += 2;
    cout << "&i = " << &i << endl;
    cout << "&j = " << &j << endl;
}

/*
- Though 'j' is initialized with the value of 'i',
  they two are independent variables.
- Changes made to 'i' doesn't affect 'j' and vice versa.
- The evidence of 'i' and 'j' being two independent
variables
  is that their addresses are distinct.
```

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

int main() {
    int i = 1;
    int &j = i;
    i++;
    j += 2;
    cout << "&i = " << &i << endl;
    cout << "&j = " << &j << endl;
}

/*
- In above program 'j' is known as reference and 'i' is
known as referent.
- The relationship between reference and referent
remains in existance till reference exists.
  It cannot be broken in between. Once reference is
released, the relation comes to an end.
```

```
int main() {
    int i = 1;
    int &j = i;
    int &k = j;

    k += 5;

    int* ptr = &j;
    *ptr = 10;
}
```

- **07 characteristics of reference**

```
int main() {
    int& j;
}

/*
- Initialization of reference is compulsory.
- It's not like pointer.
- A pointer can be defined without specificing the
address where it is
  supposed to point.
- Address can be assinged to pointer later on.
- A reference cannot be assigned referent later on.
- It has to be specified at the time of its definition.
*/
```

```
int main() {
    //int& j = nullptr;
    //int &k = 1;
    int&& r = 1;
    //int& arr[3];
    }
```

```
/*
- Reference cannot be null.
- Lvalue reference cannot refer to constant. Such
reference is called as l-value reference.
- Array of references not possible.
- What is lvalue?
  lvalue is an expression whose address is available.
*/
```

06 Reference to scalar

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

int main() {
      int i = 1;
      int j = i;
      i++;
      j += 2;
      cout << "&i = " << &i << endl;
      cout << "&j = " << &j << endl;
}

/*
- Though 'j' is initialized with the value of 'i',
  they two are independent variables.
- Changes made to 'i' doesn't affect 'j' and vice versa.
- The evidence of 'i' and 'j' being two independent
variables
  is that their addresses are distinct.
```

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

int main() {
      int i = 1;
      int &j = i;
      i++;
      j += 2;
      cout << "&i = " << &i << endl;
      cout << "&j = " << &j << endl;
}

      /*
      - In above program 'j' is known as reference and
      'i' is known as referent.
      - The relationship between reference and
      referent remains in existance till reference exists.
              It cannot be broken in between. Once
            reference is released, the relation comes to
            an end.
```

```
      int main() {
            int i = 1;
            int &j = i;
            int &k = j;

            k += 5;

            int* ptr = &j;
            *ptr = 10;
            }
```

07 characteristics of reference

```
int main() {
            int& j;
}

/*
- Initialization of reference is compulsory.
- It's not like pointer.
- A pointer can be defined without specificing
the address where it is
  supposed to point.
- Address can be assinged to pointer later on.
- A reference cannot be assigned referent later
on.
- It has to be specified at the time of its
definition.
            */
```

```
int main() {
            //int& j = nullptr;
            //int &k = 1;
            int&& r = 1;
            //int& arr[3];
            }

/*
- Reference cannot be null.
- Lvalue reference cannot refer to constant. Such
reference is called as l-value reference.
- Array of references not possible.
- What is lvalue?
  lvalue is an expression whose address is
available.
*/
```

08 const and reference

```
int main() {
            int a = 1;
            const int b = 1;

            int &ra = a;
            int& rb = b;
```

```
    const int& cra = a;
    const int& crb = b;
}

/*
-non-const refernce can refer to non-const
referent only.
-const reference can refer to const as well as
non-const referent.
    */
```

- **09 passing argument by reference**

```
void swap(int& u, int& v);

int main() {
    int a = 1, b = 2;
    swap(a, b);
}

void swap(int& u, int& v) {
    int t = u;
    u = v;
    v = t;
    }
```

- **10 returning a reference**

```
int g = 1;
int& f();

int main() {
    g = 5;
    int& h = f(); //Before call int &h = f();
and after call int &h = t;
    h = 10;
}


int&/* t */ f() {
    int& i = g;
    return i; //int &t = i;
}
```

```
int g = 1;
int& f();

int main() {
    g = 5;
    int& h = f(); //Before call int &h = f();
and after call int &h = t;
    h = 10;
}


int&/* t */ f() {
    return g; //int &t = g;
}
```

```
int g = 1;
int& f();

int main() {
    g = 5;
    int h = f(); //Before call int &h = f(); and
after call int &h = t;
    h = 10;
}


int&/* t */ f() {
    return g; //int &t = g;
}
```

```
int& f();

int main() {
    int& h = f();
    h = 10;
}

int& /* t */ f() {
    int g = 1;
    return g; // int &t = g;
}

/*
- Above program results into dangling reference
'h' in main.
*/
```

- **11 reference to the pointer**

```
void alloc(int*& pn) {
    pn = new int;
}

int main() {
    int* pa = nullptr;
    alloc(pa);
    delete pa;
    pa = nullptr;
}
```

- **12 reference to pointer**

```
#include <stdio.h>

void f(int* p) {

}

void g(int& r) {

}

int main() {

        f(NULL);
        g(NULL);
}

/*
- Reference cannot be set to null.
- If null could be one of the possible values of a parameter then
  the parameter is to be implemented as pointer.
        */
```

DAY 09

01 structure basics

```
int main() {
    /* struct defination */
    struct Point {
        int x;
        int y;
    };

    struct Point3D {
        int x;
        int y;
        int z;
    };

    /* struct variable defination and initilization*/

    Point a = { 0,0 };
    Point b = { 1,5 }, c = { -10,-3 };

    /* accesing struct members*/
    int x = a.x;
    int y = a.y;

    /* struct variable assignment*/
    a = b;

    Point3D d = { 1,-1,-1};
    //c =d; //Error variables are of dissimilar struct types
    c.x = d.x; //ok
    c.y = d.y; //ok

    /* defining pointer to struct */
    Point* pa = &a;

    /* accesing stuct members using pointyer - method 1*/

    (*pa).x = -3;
    (*pa).y = 7;

    /* accessing struct members using pointer - method 2*/
    pa->x = 8;
    pa->y = 2;


    /*dynamically allocating struct*/
    size_t count = 2;
    pa = new Point[count]{ {-1,-1}, {1,1} };
    x = pa[0].x;
    y = pa[0].y;
    delete[] pa;
    pa = nullptr;


    /* reference to struct*/
    Point& ra = a;
    ra.x = 5;
    ra.y = -5;

}

/*
- The structure 'Point' is called as user defined type (UDT).
- The 'x' and 'y' are called as data members.
- Preferrably struct is declared globally, so that it is available to
  all functions of that translation unit.
- Typically structs are declated in header file. And header file is included
  at the top of the .cpp file.
- When variables in an assignment are of same udt then the assignment can be done
  directly between them.
- When variables in an assignment are of different udts then the assignment is to be done
  member-wise.
- Use '.' operator while accessing members using variable/reference.
- Use '->' operator while accessing members using pointer.
```

02 size of structure

```
struct Point {
	int x;
	int y;
};

struct dummy {
	int n;
	char ch;
};

int main() {
	Point a = { 1,1 };
	size_t aSize = sizeof(a);
	// object passed as an argument of operator
	size_t pointSize = sizeof(Point);		// type
passed as an argument to sizeof operator
	size_t dummySize = sizeof(dummy);
	// type passed as an argument to size of operator
}

/*
- The sizeof operator can be used with an object or
type.
- Never rely on manually calculated size of a
structure/ structure variable.
- Always use sizeof operator to calculate size of
structure/Structure variable

*/
```

03 structure and function

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


struct Point {
	int x;
	int y;
};

void Print (Point P);

int main() {
	Point a = { -1,1 };
	Print(a);
}


void Print(Point p) {
	cout << "x= " << p.x << endl;
	cout << "y= " << p.y << endl;
}

/*
In this example we are passing structure argument
by value. Usually we avoid this. Because it takes
more memory and there is data redundancy.
*/
```

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

struct Point {
	int x;
	int y;
};

void Print(const Point* ppoint);

int main() {
	Point a = { -1,1 };
	Print(&a);
}


void Print(const Point* ppoint) {
	cout << "x= " << ppoint->x << endl;
	cout << "y= " << ppoint->y << endl;
}

/*
In this program we are passing structure argument
by address. This is one of the preferred methods.
We choose to declare pointer pointing to const
	structure, if the function is going to simply
	read the values. By declaring pointer to const,
	the function becomes versatile.
	In case if function's purpose is to make changes
	to original object, then we omit const
	specification
	on the pointer.
	*/
```

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

	struct Point {
			int x;
			int y;
	};

	void Print(const Point& Point );

	int main() {
			Point a = { -1,1 };
```

```
        Print(a);
    }

    void Print(const Point& Point ) {
        cout << "x= " << Point.x << endl;
        cout << "y= " << Point.y << endl;
    }

    /*
    In this program we are passing structure
    argument
    by reference. This is one of the preferred
    methods.
    We choose to declare const reference to a
    structure,
    if the function is going to simply read the values.
    By declaring const reference, the function
    becomes versatile.
    In case if function's purpose is to make changes
    to the original object, then we omit const
    specification
    on the reference.
    */
```

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

struct Point {
    int x;
    int y;
};

void Print(const Point& point);
Point /*temp*/ Offset(const Point& point, int dx, int
dy);

int main() {
    Point a = { -1,1 };
    Print(a);

    Point b = Offset(a, 1, 1);
}

void Print(const Point& point) {
    cout << "x= " << point.x << endl;
    cout << "y= " << point.y << endl;
}

Point /*temp*/ Offset(const Point& point, int dx, int
dy) {
    Point t = { 0,0 };
    t.x = point.x + dx;
    t.y = point.y + dy;
    return t;//Point temp = t
}
```

- **04 structure and array**

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

struct Person {
    char name[60];
    int age;
};

int main() {
    Person person = { "Kshitij", 35 };
    cout << "Name= " << person.name << " Age = "
<< person.age << endl;

    Person persons[] = { {"Varun", 35}, {"Shekhar",
39} };
    for (size_t i = 0; i < sizeof(person) /
sizeof(Person); i++) {
            cout << "Name: " << persons[i].name
<< "Age= " << persons[i].age << endl;

    }
    }
```

- **05 self referencial structure**

```
struct Point {
    int x;
    int y;
    Point nextPoint;
};

int main() {
    Point a;
}

/*
- A struct variable cannot be a member of
the same struct.
*/
```

```
struct Point {
    int x;
```

```
    int y;
    Point *nextPoint;
};

int main() {
    Point a;
}

/*
- A self referential data structure is
essentially a structure definition
 which includes at least one member that is
a pointer to the structure of
 its own kind.
*/
```

```
struct Point {
    int x;
    int y;
};

struct PointCollection {
    struct Node {
        Point point;
        Node* pnextNode;
    };
    Node* pheadNode;
    Node* ptailNode;
    size_t count;
};

void AddPoint(PointCollection& obj, Point
point);
void RemovePoint(PointCollection& obj,
Point point);

int main() {
    PointCollection obj = { nullptr, nullptr,
0 };

    Point u = { 1, -1 };
    AddPoint(obj, u);

    Point v = { 1, 5 };
    AddPoint(obj, v);

    Point w = { 5, -1 };
    AddPoint(obj, w);

    RemovePoint(obj, w);
    RemovePoint(obj, v);
    RemovePoint(obj, u);
}

void AddPoint(PointCollection& obj, Point
point) {
    PointCollection::Node* newNode = new
PointCollection::Node();
    newNode->point.x = point.x;
    newNode->point.y = point.y;
    newNode->pnextNode = nullptr;

    if (obj.count == 0) {
        obj.pheadNode = obj.ptailNode
= newNode;
        obj.count++;
        return;
    }

    obj.ptailNode->pnextNode = newNode;
    obj.ptailNode = newNode;
    obj.count++;
}

void RemovePoint(PointCollection& obj,
Point point) {
    if (obj.count == 0) return;

    PointCollection::Node* pprevNode =
nullptr;
    PointCollection::Node* pdelNode =
obj.pheadNode;

    /* locate the node to be deleted */
        while (pdelNode != nullptr) {

            if (pdelNode->point.x
== point.x && pdelNode->point.y ==
point.y)
                break; // found
the node
            pprevNode = pdelNode;
            pdelNode = pdelNode-
>pnextNode;
        }

    if (pdelNode != nullptr) { // node to be
deleted is located
        if (pprevNode == nullptr) { //
means pdelNode is referring to first node
            obj.pheadNode =
pdelNode->pnextNode;
        }
        else { // means pdelNode is
referring to some intermediate node or last
node.
            pprevNode->pnextNode
= pdelNode->pnextNode;
        }
```

```
			// ptailNode must point to
second last node if node to be deleted is the
last node
			if (pdelNode->pnextNode ==
nullptr) {
				obj.ptailNode =
pprevNode;
			}

			// actual deletion of the node
			pdelNode->pnextNode =
nullptr;
			delete pdelNode;
			pdelNode = nullptr;

			// reduce count by 1, since one
node is deleted from the list.
			obj.count--;
		}
}

/*
- Note struct Node is nested within struct
PointCollection.
*/
```

➢ *DAY 10*

➢ 01 C++ structure

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

struct Circle {
	int m_radius;

	void Print() {
		cout << m_radius << endl;
	}
};

int main() {
	Circle a = { 5 };

}
```

➢ 02 the this pointer

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

struct Circle {
	int m_radius;
};

int main() {
	Circle a = { 5 };
	Circle* pobj = &a;
	cout << pobj->m_radius << endl;

	Circle b = { 10 };
	pobj = &b;
	cout << pobj->m_radius << endl;

}
```

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

struct Circle {
	int m_radius;
};

void Print(Circle* pobj) {
	cout << pobj->m_radius << endl;
}

int main() {
	Circle a = { 5 };
```

```
	Print(&a);

	Circle b = { 10 };
	Print(&b);
}
```

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

struct Circle {
	int m_radius;

	void Print(Circle* pobj) {
		cout << pobj->m_radius << endl;
}

};


int main() {
	Circle a = { 5 };
	a.Print(&a);

	Circle b = { 10 };
	b.Print(&b);
}

/*
- When a function is made member function, c++
introduces 'this' pointer within the function.
*/
```

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

struct Circle {
	int m_radius;

	void Print(Circle* pobj) {
		cout << this->m_radius << endl;
	}

};


int main() {
	Circle a = { 5 };
	a.Print(&a);

	Circle b = { 10 };
	b.Print(&b);
}
```

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

struct Circle {
	int m_radius;

	void Print() {
		cout << this->m_radius << endl;
	}

};


int main() {
	Circle a = { 5 };
	a.Print();

	Circle b = { 10 };
	b.Print();
}
```

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

struct Circle {
	int m_radius;

	void Print() {
		cout << m_radius << endl;
	}

};


int main() {
	Circle a = { 5 };
	a.Print();

	Circle b = { 10 };
	b.Print();
}

/*
- Object space contains only data members.
- Member function are stored in code segment.
*/
```

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

```
struct Circle {
	int m_radius;

	void Print() {
		this = new Circle; // 'this' pointer cannot
be assigned address of another object
		cout << m_radius << endl;
	}

};


int main() {
	Circle a = { 5 };
	a.Print();

	Circle b = { 10 };
	b.Print();
}


/*
- The 'this' pointer exist only in member function and
not in global functions.
- It is set automatically by the program based upon
which object is used to call the member function.
- c++ implements 'this' pointer as constant pointer.
- Logically speaking, 'this' can be thought as first
parameter of member function.
- Note it is not a member of struct/class/object space.
- Physically, 'this' is passed via register.
- The type of 'this' pointer is same as that of the
struct/class type
*/
```

- **02a size of empty structure in c++**

```
struct Dummy {

};

int main() {
	Dummy dummy;
}


/*
- The size of empty struct is 1 byte.
- Note this byte cannot be accessed.
- It is there to indicate presense of an object.
*/
```

- **03 accesors and mutator**

```
#include<assert.h>
#include<iostream>

using namespace std;

struct Circle {

	int m_radius;

	/*accesor or getter*/
	int GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	/*mutator or setter*/

	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
			m_radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a = { 5 };
	int r = a.GetRadius();
	a.Print();
	a.SetRadius(5);
	a.Print();

	a.m_radius = 5;
	a.SetRadius(-5);
}
```

- **04 access specifiers**

```
#include<assert.h>
#include<iostream>

using namespace std;

struct Circle {

private:
```

```
	int m_radius;

public:

	/*accesor or getter*/
	int GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	/*mutator or setter*/
	void SetRadius(int radius) {
		if (radius <= 0)
			throw invalid_argument("Radius must be positive number.");
		m_radius = radius;
	}

public:
	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	//Circle a = { 5 };			//Error: m_radius is declared as private hence direct initialization not possible.
	Circle a;
	a.SetRadius(5);
	int r = a.GetRadius();
	a.Print();
	a.SetRadius(10);
	a.Print();

	//a.m_radius = 5;  //Error: m_radius is declared as private hence not accessible.
	a.SetRadius(-5);
}
```

- **05 the class**

```
class Dummy {
public:
	int n;
};

int main() {
	Dummy u;
	u.n = 1;
}

/*
- The default access of struct is public.
- The default access of class is private.
- The default access of struct or class can be overriden with the
  help of suitable access specifiers.
		*/
```

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
	int m_radius;

public:
	/*accesor or getter*/
	int GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	/*mutator or setter*/

	void SetRadius(int radius) {
		if (radius <= 0)
			throw invalid_argument("Radius must be positive number.");
		m_radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a;
	int r = a.GetRadius();
	a.Print();
	a.SetRadius(5);
	a.Print();
	a.SetRadius(-5);
	}
```

- **06 resolving name conflict between parameter and data member**

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
	int radius;

public:
	/*accesor or getter*/
	int GetRadius() {
		assert(radius > 0);
		return radius;
	}

	/*mutator or setter*/

	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
		radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a;
	int r = a.GetRadius();
	a.Print();
	a.SetRadius(5);
	a.Print();
}

/*
- When parameter and data member have same
name,
  parameter always takes precedence over data
member.
	*/
```

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {
private:
	int radius;

public:
	/*accesor or getter*/
	int GetRadius() {
		assert(radius > 0);
		return radius;
	}

	/*mutator or setter*/

	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
		this->radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a;
	int r = a.GetRadius(5);
	a.Print();
	a.SetRadius(5);
	a.Print();
}
```

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
	int radius;

public:
	/*accesor or getter*/
	int GetRadius() {
		assert(radius > 0);
		return radius;
	}

	/*mutator or setter*/
```

```
	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
		Circle::radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a;
	a.SetRadius(5);
	a.Print();
	a.SetRadius(5);
	a.Print();
}
```

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
	int m_radius;

public:
	/*accesor or getter*/
	int GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	/*mutator or setter*/

	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a;
	a.SetRadius(5);
	a.Print();
	a.SetRadius(5);
	a.Print();
}
```

- **07 constructor**

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
	int m_radius;

public:
	/*accesor or getter*/
	int GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	/*mutator or setter*/

	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a;
	//a.SetRadius(5);
	a.Print();
	a.SetRadius(10);
	a.Print();
}

/*
- At the beginning of the program, when object 'a' of
Circle is instantiated,
  it comes into existance with m_radius value set to
garbage value.
```

```
  The garbage value mostly is -ve, which is not a valid value for m_radius.
  Because 'm_radius' is marked with private access, direct assignment to it
  is prohibited. Calling SetRadius at the beginning would work but it is not a
  foolproof solution, as another developer may not call SetRadius at the begin.
- The only solution to this issue is writing constructor in the class.
        */
```

```cpp
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
    int m_radius;
public:
    Circle() {                          //Default or non parametric constructor
        m_radius = 1;
    }

public:
    /*accesor or getter*/
    int GetRadius() {
        assert(m_radius > 0);
        return m_radius;
    }

    /*mutator or setter*/

    void SetRadius(int radius) {
        if (radius <= 0)
            throw invalid_argument("Radius must be positive number.");
        m_radius = radius;
    }

    void Print() {
        cout << GetRadius() << endl;
    }
};

int main() {
    Circle a;
    a.Print();
    a.SetRadius(10);
    a.Print();
}

/*
- A constructor is a special member function of a class.
- It is called automatically at the definition of the object.
- The name of the constructor function should be same as the class name.
- No return type has to be specified on constructor function.
- Specifing return type on the constructor results into compilation error.
- When a class is written without any constructor, compiler provides default constructor.
- This constructor doesn't do anything.
- Writing a constructor of any form (such as default or parametric or copy) prohibits compiler from
  adding default constructor on its own.
- Constructor provides us opportunity to initialize the object data members.
- If we miss this opportunity i.e. we didnot write initialization code in
  constructor then the object data members will remain uninitialized.
        */
```

```cpp
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
    int m_radius;
public:
    Circle() {
        m_radius = 1;
    }

public:
    /*accesor or getter*/
    int GetRadius() {
        assert(m_radius > 0);
        return m_radius;
    }

    /*mutator or setter*/

    void SetRadius(int radius) {
        if (radius <= 0)
```

```
            throw
invalid_argument("Radius must be positive
number.");
            m_radius = radius;
    }

    void Print() {
            cout << GetRadius() << endl;
    }
};

int main() {
    Circle a;
    a.Circle();        //Its a error to call constructor
explicitly
    a.Print();
}

/*
- For any object constructor is called once and only
once.
  However, it is called for every object.
- The constructor cannot be called explicitly on an
object.
- Attempting to call constructor explicitly results
into compilation error.
        */
```

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
    int m_radius;
public:
    Circle(int n) {                    //Parametric
constructor
            SetRadius(n);
    }

public:
    /*accesor or getter*/
    int GetRadius() {
            assert(m_radius > 0);
            return m_radius;
    }

    /*mutator or setter*/

    void SetRadius(int radius) {
            if (radius <= 0)
            throw
invalid_argument("Radius must be positive
number.");
            m_radius = radius;
    }

    void Print() {
            cout << GetRadius() << endl;
    }
};

int main() {
    Circle a(5); //this syntax works for single and
multi parametric constructor

    Circle b = 10; // this syntax works for single
parametric constructor only.

    Circle c = { 5 }; // this syntax works for single
and multi parametric constructor
}
```

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
    int m_radius;
public:
    Circle() {
            m_radius = 1;            //Default
constructor
    }


    Circle(int n) {                    //Parametric
constructor
            SetRadius(n);
    }

public:
    /*accesor or getter*/
    int GetRadius() {
            assert(m_radius > 0);
            return m_radius;
    }

    /*mutator or setter*/
```

```
	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a; // Invokes default constructor
	a.Print();

	Circle b(5); // Invokes parametric constructor
	b.Print();

	Circle c = 10; // Invokes parametric constructor,
if it is single parametric
}

/*
Constructor overloading (function overloading of
constructor) is allowed.
		*/
```

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
	int m_radius;

	Circle(int n = 1) {
	//Parametric constructor
		SetRadius(n);
	}

public:
	/*accesor or getter*/
	int GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	/*mutator or setter*/
```

```
	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}
};

int main() {
	Circle a; // Invokes default constructor
	a.Print();

	Circle b(5); // Invokes parametric constructor
	b.Print();

	Circle c = 10; // Invokes parametric constructor,
if it is single parametric
}

/*
Constructor overloading (function overloading of
constructor) is allowed.
```

```
#include<iostream>
#include<stdexcept>
#include<assert.h>
using namespace std;

class Circle {
private:
	int m_radius;

public:
	Circle(int radius = 1) {// this works as a default
as well as a paramteric constructor
		SetRadius(radius);

	}
public:
	/*accessor or getter*/
	int GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}
```

```
        /*mutator or setter*/
        void SetRadius(int radius) {
            if (radius <= 0)
                throw
invalid_argument("Radius must be positive");
        m_radius = radius;
        }
public:
    void Print() {
        cout << GetRadius() << endl;
    }
    };

    int main() {
        Circle u; // Check if constructor is called
for 'u'?
        Circle* pu = &u; // Check if constructor
is called for pointer?
        Circle& ru = u; // Check if constructor is
called for reference?
    }

    /*
    - Constructor is not called for pointer and
reference.
    - Since constructor is not called, the destructor is
also not called
      for pointer and reference.

    - Why constructor is not called for pointer?
      Remember, pointer is not an object. It points to
an object.
      Hence since its not an object in itself,
constructor is not called.

    - Why constructor is not called for reference?
      Reference refers to referent. Since referent is
already constructed there is
      no need to call constructor for the reference.
    */
```

```
#include<iostream>
#include<stdexcept>
#include<assert.h>
using namespace std;

class Circle {
private:
    int m_radius;
private:
    Circle(int radius = 1) {// this works as a default
as well as a paramteric constructor
        SetRadius(radius);

    }
public:
    /*accessor or getter*/
    int GetRadius() {
        assert(m_radius > 0);
        return m_radius;
    }


    /*mutator or setter*/
    void SetRadius(int radius) {
        if (radius <= 0)
            throw
invalid_argument("Radius must be positive");
        m_radius = radius;
    }
public:
    void Print() {
        cout << GetRadius() << endl;
    }
};

int main() {
    Circle u; // Check if constructor is called for 'u'?
    Circle* pu = &u; // Check if constructor is called
for pointer?
    Circle& ru = u; // Check if constructor is called
for reference?
}


/*
- if constructor is declared private then that class can
not be instatiated using constructor.
        */
```

➢ **08 operations**

```
#include<assert.h>
#include<iostream>

using namespace std;

class Circle {

private:
	int m_radius;
public:


	Circle(int n=1) {
	//Parametric constructor
		SetRadius(n);
	}

public:
	/*accesor or getter*/
	int GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	/*mutator or setter*/

	void SetRadius(int radius) {
		if (radius <= 0)
			throw
invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Print() {
		cout << GetRadius() << endl;
	}

	double GetArea() {
		int radius = GetRadius();
		return 3.14 * radius * radius;
	}
};

int main() {
	Circle a; // Invokes default constructor
	a.Print();
	double area = a.GetArea();

	Circle b(5); // Invokes parametric constructor
	b.Print();
	area = b.GetArea();

	Circle c = 10; // Invokes parametric constructor,
if it is single parametric
	area = c.GetArea();
}
```

➢ **09 two ways of defining member functions**

➢ **01 member functions defined inside the class**

Circle.h

```
#pragma once
#include<assert.h>
#include<iostream>

class Circle {

private:
	int m_radius;
public:


	Circle(int n = 1) {
	//Parametric constructor
		SetRadius(n);
	}

public:
	/*accesor or getter*/
	int GetRadius() {
		//assert(m_radius > 0);
		return m_radius;
	}

	/*mutator or setter*/

	void SetRadius(int radius) {
		if (radius <= 0)
			throw
std::invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Print() {
		std::cout << GetRadius() << std::endl;
	}

	double GetArea() {
		int radius = GetRadius();
		return 3.14 * radius * radius;
	}
```

```
};
```

Source01.cpp

```
#include "Circle.h"

int main() {
	Circle a; // Invokes default constructor
	a.Print();
	double area = a.GetArea();

	Circle b(5); // Invokes parametric constructor
	b.Print();
	area = b.GetArea();

	Circle c = 10; // Invokes parametric constructor,
if it is single parametric
	area = c.GetArea();
}

/*
-When member funvtions define in the class, they
are inline by default.
	*/
```

- **02 member functions defined outside the class**

Circle.cpp

```
#include<assert.h>
#include<iostream>
#include<stdexcept>
#include "Circle.h"
using namespace std;


	Circle::Circle(int n ) {
	//Parametric constructor
		SetRadius(n);
	}

	int Circle::GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	void Circle::SetRadius(int radius) {
		if (radius <= 0)
			throw
std::invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Circle::Print() {
		std::cout << GetRadius() << std::endl;
	}

	double Circle::GetArea() {
		int radius = GetRadius();
		return 3.14 * radius * radius;
	}
```

Circle.h

```
#pragma once
#include<assert.h>
#include<iostream>

class Circle {

private:
	int m_radius;
public:
	Circle(int n = 1);
public:
	int GetRadius();
	void SetRadius(int radius);
	void Print();
	double GetArea();
		};
```

Source01.cpp

```
#include "Circle.h"

int main() {
	Circle a; // Invokes default constructor
	a.Print();
	double area = a.GetArea();

	Circle b(5); // Invokes parametric constructor
	b.Print();
	area = b.GetArea();

	Circle c = 10; // Invokes parametric constructor,
if it is single parametric
	area = c.GetArea();
}

/*
- when member functions are defined outside the
class, they are not inline by default
*/
```

- **10 constant member functions**
  - **01 calling non const member function on const object**

Circle.cpp

```
#include<assert.h>
#include<iostream>
#include<stdexcept>
#include "Circle.h"
using namespace std;


	Circle::Circle(int n ) {
	//Parametric constructor
		SetRadius(n);
	}

	int Circle::GetRadius() {
		assert(m_radius > 0);
		return m_radius;
	}

	void Circle::SetRadius(int radius) {
		if (radius <= 0)
			throw
std::invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Circle::Print() {
		std::cout << GetRadius() << std::endl;
	}

	double Circle::GetArea() {
		int radius = GetRadius();
		return 3.14 * radius * radius;
	}
```

Circle.h

```
#pragma once
#include<assert.h>
#include<iostream>

class Circle {

private:
	int m_radius;
public:
	Circle(int n = 1);
public:
	int GetRadius();
	void SetRadius(int radius);
	void Print();
	double GetArea();
		};
```

Source01.cpp

```
#include "Circle.h"

int main() {
	const Circle a =5; // Invokes default constructor
	int r = a.GetRadius();
	a.Print();
	double area = a.GetArea();
}

/*
- Non const function cannot be called on const
object.
- Presently, 'GetRadius', 'Print' and 'GetArea' are non
const member functions.
		*/
```

- **02 declaring member function as const member function**

Circle.cpp

```
#include<assert.h>
#include<iostream>
#include<stdexcept>
#include "Circle.h"
using namespace std;


	Circle::Circle(int n ) {
	//Parametric constructor
		SetRadius(n);
	}

	int Circle::GetRadius() const {
		assert(m_radius > 0);
		return m_radius;
	}

	void Circle::SetRadius(int radius) {
		if (radius <= 0)
			throw
std::invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}
```

```cpp
void Circle::Print() const {
        std::cout << GetRadius() << std::endl;
}

double Circle::GetArea() const{
        int radius = GetRadius();
        return 3.14 * radius * radius;
}
```

Circle.h

```cpp
##pragma once
#include<assert.h>
#include<iostream>

class Circle {

private:
    int m_radius;
public:
    Circle(int n = 1);
public:
    int GetRadius() const;
    void SetRadius(int radius) ;
    void Print() const;
    double GetArea() const;
};

/*
- Declaring a member function with the const keyword specifies that
  the function is a "read-only" function i.e. it does not help modify
  state of the object for which it is called.

- To declare a constant member function, place the const keyword after
  the closing parenthesis of the argument list.
  The const keyword is required in both the declaration and the definition.

- A const function can be called on const and non-const object.
- A non-const function can be called only on non-const object.
*/
```

Source01.cpp

```cpp
#include "Circle.h"

int main() {
    const Circle a =5; // Invokes default constructor
    int r = a.GetRadius();
    a.Print();
    double area = a.GetArea();
}

const member functions.
        */
```

- **03 characteristics of const member function**

```cpp
class Dummy {
public:
    void F() ;
    void G() const;
};

void Dummy::F(){}

void Dummy::G()const {}

int main() {
    Dummy u;
    u.F();
    u.G();

    const Dummy v;
    v.F();
    v.G();
}
/*
- A const function can be called on const and non-const object.
- A non-const function can be called only on non-const object.
- Thus, const member functions are versatile functions than non-const member functions.
- While declaring new class, try to declare member functions const wherever possible.
  If the purpose of the function is to allow mutation to data members then
  such function cannot be declared as const member function.
*/
```

```cpp
int g_data;

class Dummy {
public:
	void F() const;
private:
	int m_data;
};

void Dummy::F() const {
	int data = 0;
	data = 1;
	g_data = 1;
	m_data = 1;
}

int main() {}

/*
- local variables can be modified within const function.
- global variables can be modified within const function.
- data members cannot be modified within const function.
*/
```

```cpp
class Dummy {
public:
	void F() const;
	static void G() const;
};

void Dummy::F() const {}

void Dummy::G() {}

int main() const {}

/*
- Global and static member functions cannot be constant functions,
  since they lack the presence of 'this' pointer.
- Only non static member functions can be constant functions.
*/
```

```cpp
class Dummy {
public:
	void F();
	void G() const;
public:
	void H();
	void K() const;
};

void Dummy::F() {}

void Dummy::G() const {}

void Dummy::H() {
	F();
	G();
}

void Dummy::K() const {
	F();
	G();
}

int main() {}

/*
-A non - const member function can call const as well as non - const member functions.
- A const member function can call only const member functions.
* /
```

```cpp
class Dummy {
public:
	void F() ;
	void F() const;
};

void Dummy::F()  {}

void Dummy::F() const {}

int main()  {
	Dummy u;
	u.F();

	const Dummy v;
	v.F();
}

/*
- Function overloading is possible between const and non-const non-static member functions.
*/
```

- **11 mutable**

Circle.cpp

```
#include<assert.h>
#include<iostream>
#include<stdexcept>
#include "Circle.h"
using namespace std;


	Circle::Circle(int n )  {
	//Parametric constructor
		SetRadius(n);
	}

	int Circle::GetRadius() const {
		assert(m_radius > 0);
		return m_radius;
	}

	void Circle::SetRadius(int radius) {
		if (radius <= 0)
			throw
std::invalid_argument("Radius must be positive
number.");
		m_radius = radius;
		m_area = -1;
	}

	void Circle::Print() const {
		std::cout << GetRadius() << std::endl;
	}

	double Circle::GetArea() const{
		int radius = GetRadius();
		if(m_area < 0)
		return 3.14 * radius * radius;
	}
```

Circle.h

```
#pragma once
#include<assert.h>
#include<iostream>

class Circle {

private:
	int m_radius;
	mutable double m_area;
public:
	Circle(int n = 1);
public:
	int GetRadius() const;
	void SetRadius(int radius) ;
	void Print() const;
	double GetArea() const;
};
```

Source01.cpp

```
#include "Circle.h"

int main() {
	Circle a = 5;
	double area = 0.0;

	area = a.GetArea();
	area = a.GetArea();

	a.SetRadius(10);
	area = a.GetArea();
	area = a.GetArea();
}
```

DAY 09

- **01 object initialization**
  - ***01 using body of a constructor***

Integer.cpp

```
#include "Integer.h"

Integer::Integer(int i) {
	m_i = i;
}

int Integer::Get() const{
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}
```

Integer.h

```
#pragma once

class Integer {
public:
	Integer(int i = 0);
public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};
```

Source.cpp

```
#include<assert.h>
#include "Integer.h"

int main() {
	Integer u(5);
	assert(u.Get() == 5);

	u.Set(10);
	assert(u.Get() == 10);
    }
```

- ***02 using initilisation list***

Integer.cpp

```
#include "Integer.h"

Integer::Integer(int i): m_i(i) {}

int Integer::Get() const{
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}
```

Integer.h

```
#pragma once

class Integer {
public:
	Integer(int i = 0);
public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};
```

Source.cpp

```
#include<assert.h>
#include "Integer.h"

int main() {
	Integer u(5);
	assert(u.Get() == 5);

	u.Set(10);
	assert(u.Get() == 10);
}
```

- ***03 more on initialization list***

```
#include <assert.h>

int F();

class Integer {
public:
	Integer();
	Integer(int i);
public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};

Integer::Integer() : m_i(F()) {}		//note we are calling function 'F'  to initialise m_i
Integer::Integer(int i) : m_i(i) {}

int Integer::Get() const {
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}

int F() { return 1; }


int main() {
	Integer u;
	assert(u.Get() == 1);
```

```
}
```

```
#include <assert.h>

void F();

class Integer {
public:
	Integer();
	Integer(int i);
public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};

Integer::Integer() : m_i(F()) {}		//note we cannot call function 'F' since its not returning value
Integer::Integer(int i) : m_i(i) {}

int Integer::Get() const {
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}

void F() {}


int main() {
	Integer u;
	assert(u.Get() == 1);
}
```

```
#include <assert.h>


class Integer {
public:
	Integer();
	Integer(int i);
public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};

Integer::Integer() : Set(i) {}			//note data member cannot be skipped in initialization list
Integer::Integer(int i) : m_i(i) {}

int Integer::Get() const {
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}


int main() {
	Integer u;
	assert(u.Get() == 1);
}
```

```
#include <assert.h>


class Integer {
public:
	Integer();
	Integer(int i);
public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};

Integer::Integer() : m_i(0) {}
Integer::Integer(int i) : m_i(i) {}

int Integer::Get() const {
	return m_i;
}

void Integer::Set(int i):m_i(i) {
	m_i = i;
}


int main() {
	Integer u;
	assert(u.Get() == 1);
}

/*
-Intitialization list can be used only with constructor.
-It cannot be used with any other member function.
*/
```

- **02 Overloading arithmetic operators**
  - ***01 base code***

Integer.cpp

```
#include "Integer.h"

Integer::Integer(int i) {
	m_i = i;
}

int Integer::Get() const{
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}
```

Integer.h

```
#pragma once

class Integer {
public:
	Integer(int i = 0);
public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};
```

Source.cpp

```
#include<assert.h>
#include "Integer.h"

int main() {
	Integer u(5), v(10), w;
	w = u + v;
}

/*
- Compiler fails to evaluate w = u + v, because u, v, w
are objects of Integer.
- Integer is a UDT (user defined type).
- Compiler knows nothing about which member of u is
to be added with
  which member of v and assign result to which member
of w.
- Because of this, compiler throws error on u + v
expression.
```

- ***02 overloading operator as a member function***

Integer.cpp

```
#include "Integer.h"

Integer::Integer(int i) {
	m_i = i;
}

int Integer::Get() const{
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}

Integer Integer::operator+(const Integer& obj) {
	Integer result;
	result.m_i = m_i + obj.m_i;
	return result;
}
```

Integer.h

```
#pragma once

class Integer {
public:
	Integer(int i = 0);
public:
	int Get() const;
	void Set(int i);
public:
	Integer operator+(const Integer& obj);
private:
	int m_i;
};
```

Source.cpp

```
#include<assert.h>
#include "Integer.h"

int main() {
	Integer u(5), v(10), w;
	w = u + v; // w=u+v; executed as w =
u.operator+ (v)
}

/*
```

Operator overloading is a feature of c++. With this feature, we can provide
implementation for execution of respective operation in the context of
our objects.

Operator overloading doesn't permit changes to operator precedence table.
Hence we cannot add new operators i.e. we can overload existing operators
only. There are few operators such as ::, ., sizeof, typeid, ?: which cannot
be overloaded. Most of the operators can be implemented either in member
function format or global function format. There are few operators
however for ex. (), [], (casting) etc. have to be implemented in member
function format only. Wherever global function format is supported, it's
recommended to use global function format.
*/

- ***03 overloading operator as a global function***

Integer.cpp

```
#include "Integer.h"

Integer::Integer(int i) {
	m_i = i;
}

int Integer::Get() const{
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}


Integer operator+(const Integer& lobj, const Integer& robj) {
	Integer result;
	result.Set(lobj.Get() + robj.Get());
	return result;
    }
```

Integer.h

```
#pragma once

class Integer {
public:
	Integer(int i = 0);
public:
	int Get() const;
	void Set(int i);


private:
	int m_i;
};
Integer operator+(const Integer& lobj, const Integer& robj);
```

Source.cpp

```
#include<assert.h>
#include "Integer.h"

int main() {
	Integer u(5), v(10), w;
	w = u + v; // w=u+v; executed as w =
}

/*
- Compiler fails to evaluate w = u + v, because u, v, w are objects of Integer.
- Integer is a UDT (user defined type).
- Compiler knows nothing about which member of u is to be added with
  which member of v and assign result to which member of w.
- Because of this, compiler throws error on
u + v expression.
*/
```

03 copy constructor

Integer.cpp

```cpp
#include "Integer.h"

Integer::Integer(int i) {
	m_i = i;
}

int Integer::Get() const{
	return m_i;
}

Integer::Integer(const Integer& obj):m_i(obj.m_i){}

void Integer::Set(int i) {
	m_i = i;
}


Integer operator+(const Integer& lobj, const Integer&
robj) {
	Integer result;
	result.Set(lobj.Get() + robj.Get());
	return result;
    }
```

Integer.h

```cpp
#pragma once

class Integer {
public:
	Integer(int i = 0);
	Integer(const Integer& obj);

public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};
```

Source.cpp

```cpp
#include<assert.h>
#include "Integer.h"

int main() {
	Integer a(1);
	Integer b = a;
	assert(a.Get() == b.Get());
}
```

```
/*
- Copy constructor always exist in the class.
- Compiler supplied copy constructor do byte by byte
copy.
*/  which member of v and assign result to which
member of w.
- Because of this, compiler throws error on
u + v expression.
*/
```

04 destructor

Integer.cpp

```cpp
#include "Integer.h"

Integer::Integer(int i) {
	m_i = i;
}

Integer::~Integer() {

}

int Integer::Get() const{
	return m_i;
}

void Integer::Set(int i) {
	m_i = i;
}
```

Integer.h

```cpp
#pragma once

class Integer {
public:
	Integer(int i = 0);
	~Integer();
public:
	int Get() const;
	void Set(int i);
private:
	int m_i;
};
```

Source.cpp

```cpp
#include<assert.h>
#include "Integer.h"
```

```
int main() {
	Integer a;
}

/*
- The destructor is a special member function, which is called automatically
  before object is about to be released.
- Remember constructor never allocates object space and
  destructor never releases object space.
- Destructor always exist in the class.
- If we omit writing destructor then compiler supplies the destructor.
- Compiler suppled destructor doesn't do do anything.
*/
```

```
class Dummy {
public:
	~Dummy();
};

Dummy::~Dummy() {}

int main() {
	Dummy u;
	u.~Dummy();
}

/*
- Explicit call to a destructor is possible.
- Note constructor cannot be called explicitly.
- Usually we do not call destructor explicitly.
*/
```

```
class Dummy {
public:
	~Dummy();
	~Dummy(int n);
};

Dummy::~Dummy() {}

Dummy::~Dummy(int n) {}

int main() {
	Dummy obj;
}

/*
- Parameterized destructor is not allowed hence
  destructor overloading is not possible.
*/
```

- **05 using and releasing resources**

Integer.cpp

```
#include<assert.h>
#include "Integer.h"

Integer::Integer(int i) : m_pi(new int(i)) {

}
Integer::~Integer() {
	delete m_pi;
	m_pi = nullptr;
}

int Integer::Get() const{
	assert(m_pi != nullptr);
	return *m_pi;
}

void Integer::Set(int i) {
	assert(m_pi != nullptr);
	*m_pi = i;
}
```

Integer.h

```
#pragma once

class Integer {
public:
	Integer(int i = 0);
	~Integer();
public:
	int Get() const;
	void Set(int i);
private:
	int *m_pi;
};
```

Source.cpp

```
#include<assert.h>
#include<crtdbg.h>
#include "Integer.h"

int main() {
	{
		Integer a;
		a.Set(5);
		assert(a.Get() == 5);
		int retval = a.Get();
	}
	_CrtDumpMemoryLeaks();
```

```
}

/*
- Resource is some aid that object would require to
perform its own functions.
- During lifetime of an object, it may acquire resource at
any point of time.
- One possible case could be at the begining of its
lifetime i.e. in the constructor.
- Note it is not compulsory for an object to acquire
resource in the constructor.
- It can acquire in any other function as well.
- Its obligation on object to release acquired resources.
- Again it can be performed at any point of time during
lifetime of an object.
- It must however be completed before object is released.
- If not done the resource would get leaked.
- The last chance to release resources is in destructor,
though it can be
  released in any other member function.
*/
```

- **06 shallow copy**

Integer.cpp

```
#include<assert.h>
#include "Integer.h"

Integer::Integer(int i) : m_pi(new int(i)) {

}
Integer::~Integer() {
	delete m_pi;
	m_pi = nullptr;
}

int Integer::Get() const{
	assert(m_pi != nullptr);
	return *m_pi;
}

void Integer::Set(int i) {
	assert(m_pi != nullptr);
	*m_pi = i;
}
```

Integer.h

```
#pragma once

class Integer {
public:
	Integer(int i = 0);
	~Integer();
public:
	int Get() const;
	void Set(int i);
private:
	int *m_pi;
};
```

Source.cpp

```
#include<assert.h>
#include "Integer.h"

int main() {
	Integer a = 5;
	Integer b = a;

	//assert(a.Get()==b.Get() );

	//Integer c(10);
	//b = c;

	//assert(b.Get() == c.Get());
}
```

- **07 deep copy**

Integer.cpp

```
#include<assert.h>
#include "Integer.h"

Integer::Integer(int i) : m_pi(new int(i)) {}

Integer::Integer(const Integer& obj): m_pi(new
int(*obj.m_pi)) {
}


Integer::~Integer() {
	delete m_pi;
	m_pi = nullptr;
}

int Integer::Get() const{
	assert(m_pi != nullptr);
	return *m_pi;
}

void Integer::Set(int i) {
```

```
        assert(m_pi != nullptr);
        *m_pi = i;
}

Integer Integer::operator=(const Integer& obj) {
        if (this == &obj)
                return *this;
        *m_pi = *obj.m_pi;
        return *this;
}
}
```

Integer.h

```
#pragma once

class Integer {
public:
        Integer(int i = 0);
        Integer(const Integer& obj);
        ~Integer();
public:
        int Get() const;
        void Set(int i);
public:
        Integer operator=(const Integer& obj);
private:
        int *m_pi;
};
```

Source.cpp

```
#include<assert.h>
#include "Integer.h"

int main() {
        Integer a = 5;
        Integer b = a;

        assert(a.Get()==b.Get() );

        Integer c(10);
        b = c;

        assert(b.Get() == c.Get());
}

/*
- If class contains pointer member then it is strongly recommened to have
  deep copy implementation of copy constructor and copy assignment operator.
- Note assignment operator has to be implemented as a member function.
  Global function format for assignment operator is NOT SUPPORTED.
- Copy constructor, assignment operator and destructor forms trio.
- What it means is? if we implement one of them, mostly we need implementation other two.
- Remember copy constructor, assignment operator and destructor always exists in the class whether we write or not.
*/
```

- **08 namespace**

```
namespace MFC {
        namespace FileServices {
                class CFile {};

                class CMemFile {};
        }

        namespace Exceptions {
                class CException {};

                class CDaoException {};

                class COleException {};
        }

        namespace Arrays {
                class CArray {};

                class CByteArray {};
        }
}

int main() {
        MFC::Arrays::CArray a;
        MFC::Exceptions::CDaoException b;
}

/*
- There are two ways, symbols of an application can be grouped viz. logically and physically.
- Namespaces are used to group symbols logically.
- Libraries are used to group symbols physically.
- Nesting of namespaces is possible.
- To refer symbol belonging to a namespace, a reference of namespace has to be mentioned alongwith symbol name.
*/
```

```cpp
namespace MFC {
	namespace FileServices {
		class CFile {};

		class CMemFile {};
	}

	namespace Exceptions {
		class CException {};

		class CDaoException {};

		class COleException {};
	}

	namespace Arrays {
		class CArray {};

		class CByteArray {};
	}
}

using namespace MFC::Arrays;
using namespace MFC::Exceptions;

int main() {
	CArray a;
	CDaoException b;
}

/*
- "using namespace" can be used to avoid typing fully qualified name of the symbol.
*/
```

```cpp
namespace MFC {
	namespace FileServices {
		class CFile {};

		class CMemFile {};
	}

	namespace Exceptions {
		class CException {};

		class CDaoException {};

		class COleException {};
	}

	namespace Arrays {
		class CArray {};

		class CByteArray {};
	}
}

using CArray = MFC::Arrays::CArray;

int main() {
	CArray a;
	CByteArray b;
}
```

➢ *DAY 13*

➢ 01 static member

Circle.cpp

```cpp
#include<assert.h>
#include<iostream>
#include<stdexcept>
#include "Circle.h"
using namespace std;


	Circle::Circle(int n ) {
	//Parametric constructor
		SetRadius(n);
	}

	int Circle::GetRadius() const {
		assert(m_radius > 0);
		return m_radius;
	}

	void Circle::SetRadius(int radius) {
		if (radius <= 0)
			throw
std::invalid_argument("Radius must be positive number.");
		m_radius = radius;
	}

	void Circle::Print()const {
		std::cout << GetRadius() << std::endl;
	}

	// this member function uses object's radius
	double Circle::GetArea() const {
		int radius = GetRadius();
		return 3.14 * radius * radius;
	}
```

```
	// this member function doesn't uses object's radius
	double Circle::GetArea(int radius)  {
		return 3.14 * radius * radius;
	}
```

Circle.h

```
#pragma once
#include<assert.h>
#include<iostream>

class Circle {

private:
	int m_radius;
public:
	Circle(int n = 1);
public:
	int GetRadius() const;
	void SetRadius(int radius);
	void Print() const;
	double GetArea() const;
	static double GetArea(int radius);
	};
```

Source.cpp

```
#include "Circle.h"

int main() {
	Circle a(10);
	double area = a.GetArea();
	//area = circle::GetArea(); //It's a error to call non-static member function using only class name.

	area = a.GetArea(20);
	area = Circle::GetArea(30);

}

/*
- Non-static member function cannot be called using only class name
  It can be called using instance only.
- Static member function can be called using both instance and class name.
- Static member function lack presence of 'this' pointer.
  Hence static member function cannot be constant member function.
- Note 'static' is to be mentioned in the member function declaration only.
  Mentioning same in the definition throws syntax error.
- Non-static methods are also called as instance methods.
- Static methods are also called as class methods.
*/
```

- **02 more on static member functions**

```
class Dummy {
public:
	void F();
	static void G();
public:
	void H();
	static void K();
};

void Dummy::F() {}

void Dummy:: G(){}

void Dummy::H() {
	F();
	G();
}

void Dummy::K() {
	F();
	G();
}

/*
- A non static member function can call both static and non-static member functions.
- A static member function can call only static member function.
*/
```

- **02 static data member**

Circle.cpp

```
#include<assert.h>
#include<iostream>
#include<stdexcept>
#include "Circle.h"
using namespace std;

//Static data member defined here
//initilization is optional. If not initilized, it is set to zero.
const double Circle::m_PI = 3.14;

	Circle::Circle(int n ) {
	//Parametric constructor
		SetRadius(n);
	}

	int Circle::GetRadius() const {
		assert(m_radius > 0);
		return m_radius;
```

```
	}

	void Circle::SetRadius(int radius) {
		if (radius <= 0)
			throw
std::invalid_argument("Radius must be positive
number.");
		m_radius = radius;
	}

	void Circle::Print()const {
		std::cout << GetRadius() << std::endl;
	}

	// this member function uses object's radius
	double Circle::GetArea() const {
		int radius = GetRadius();
		return m_PI * radius * radius;
	}

	// this member function doesn't uses object's
radius
	 double Circle::GetArea(int radius)  {
		return m_PI * radius * radius;
	}
```

Circle.h

```
#pragma once
#include<assert.h>
#include<iostream>

class Circle {

private:
	int m_radius;
	static const double m_PI;
public:
	Circle(int n = 1);
public:
	int GetRadius() const;
	void SetRadius(int radius);
	void Print() const;
	double GetArea() const;
	static double GetArea(int radius);
	  };
```

Source.cpp

```
#include "Circle.h"

int main() {
	Circle a(10);
	Circle b(20);
	double area = a.GetArea();
	//area = circle::GetArea(); //It's a error to call
non-static member function using only class name.

	area = a.GetArea(20);
	area = Circle::GetArea(30);

}

/*
- If a data member is declared as static data member then
  it needs to be defined in the implementation soure file
of
  respective class.
- If not defined, it results into linking error.
- Static data members are stored in static space.
  Non-static data members are stored in object space.
- Static data members need to be implemented in
implementation
  file. If not implemented it results into linking error.
- Static data members are also known as class variables.
- Non-static data members are also known as instance
variables.
*/
```

➢ DAY 14

➢ 01 inheritance syntax

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

//Base class can also be called as
//Parent or super or general

class Base {
public:
	void Print() {
		cout << "From Base::Print" << endl;
	}
};

//Derived class can also be called as
//child or sub or special

class Derived : public Base {

};

int main() {
	Derived d;
	d.Print();
}

//Inheritance is also knwon as 'is-a' or
//general-special relationship.


//composition/aggregation is also known as

	//'Has-a' or whole-part relationship.
```

➢ 02 constructor destructor order in inheritance

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

class Base {
public:
	Base () {
		cout << "From Base constructor" <<
endl;
	}
	~Base() {
		cout << "From Base constructor" <<
endl;
	}

};

class Derived : public Base {
public:
	Derived() {
		cout << "From Derived constructor" <<
endl;
	}
	~Derived() {
		cout << "From Derived constructor" <<
endl;
	}


};

int main() {
	Derived d;
}

/*
This program illustrates order of constructor and
destructor that
happens between Base and Derived class.

When instantiated object of Derived i.e. 'd', we observed
it was constructor
of 'Base' class that got executed first and then the
constructor of 'Derived'.

The destructor order is always reverse of constructor
order.

So at the end of the program, the destructor of 'Derived'
was executed first
and then the destructor of 'Base' class.
*/
```

➢ 03 passing arguments to base constructor from derived constructor

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

class Base {
public:
	Base (int i): m_i(i) {
	}
	~Base() {
	}
protected:
	int m_i;

};

class Derived : public Base {
public:
	Derived(int i, int j): Base(i), m_j(j) {
	}
```

```
	~Derived() {
	}
public:
	void Print() {
		// protected member m_i is accessible to
derived class
		cout << m_i << endl;
		cout << m_j << endl;
	}
private:
	int m_j;
};

int main() {
	Derived d(5,10);
	//d.m_i = 1; // protected member is nota
accesible to non member
	d.print();
}

/*
- Please note private data members of the base class are
inherited in
  derived class. They are however not accessible in
derived class.
	*/
```

➢ **04 protected access specifier**

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

class Base {
public:
	Base (int i): m_i(i) {
		cout << "From Base constructor" <<
endl;
	}
	~Base() {
		cout << "From Base constructor" <<
endl;
	}
private:
	int m_i;

};

class Derived : public Base {
public:
	Derived(int i, int j): Base(i), m_j(j) {
		cout << "From Derived constructor" <<
endl;
	}
	~Derived() {
		cout << "From Derived constructor" <<
endl;
	}
private:
	int m_j;
};

int main() {
	Derived d(1,4);
}

/*

	We avoid declaring data members protected, as
that would unable
	us to make changes to the data members in
future. We may declare
	member functions protected.
*/
```

➢ **05 how to imvoke base members**

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

class Base {
public:
	Base (int i): m_i(i) {
	}
	~Base() {
	}
public:
	void Print() {
		cout << m_i << endl;
	}

private:
	int m_i;

};

class Derived : public Base {
public:
	Derived(int i, int j): Base(i), m_j(j) {
	}
	~Derived() {
	}

public:
	void Print() {
		Base::Print();
		cout << m_j << endl;
	}
private:
```

```
	int m_j;
};

int main() {
	Derived d(1,4);
	d.Print();		// calls Print of Derived class
	d.Base::Print(); // calls Print of Base class.
}

/*
- The Derived::Print is overriding Base::Print.
- This is called function overriding.
*/
```

- **06 inheritance assignments**

```
class Base {
public:
	void f() { m_i = 1; }
	void g() { m_i = 5; }
private:
	int m_i;
};

class Derived : public Base {
public:
	void h() { m_j = 10; }
	void k() { m_j = 50; }
private:
	int m_j;
};

int main() {
	Base base;
	Derived derived;

	Base* pbase = nullptr;
	Derived* pderived = nullptr;

	pbase = &base;

	pderived = &derived;

	pbase = &derived;   //Most Imporant
Assignment*************

	pderived = &base;
}

/*
- A base class pointer can point to objects of itself and of
derived.
- A derived class pointer however can point to its own
object only i.e
 it cannot point to base object.
```

- **07 static binding**

```
class Base {
public:
	void f() {

	}
};

class Derived : public Base {
public:
	void f() {

	}
};

int main() {
	Base base;
	Derived derived;

	Base* pbase = nullptr;
	Derived* pderived = nullptr;

	pbase = &base;
	pbase->f(); //Base::f()

	pderived = &derived;
	pderived->f(); // derived::f()

	pbase = &derived;
	pbase -> f(); // Base:: f();
}
```

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

class Base {
public:
	void f() {
		cout << "From Base::f" << endl;
	}
};

class Derived : public Base {
public:
	void f() {
		Base::f();
		cout << "From Derived::f" << endl;
	}
};

int main() {
```

No, a pointer to a derived class cannot point to an object of a base class in C++. This is because the derived class is expected to have all the properties and behaviors of the base class, plus additional ones. Therefore, a derived class pointer assumes that the object it points to contains the derived class's members, which won't be true for a base class object

```
	Derived derived;
	Base* pbase = &derived;
	pbase->f(); // Base::f
	derived.f(); // Derived::f
}

/*
- There is an issue with the above setup.
- The context referred in above code i.e. derived object is same.
  The behaviour exhibited by the context is however inconsistent when approached directly and indirectly.
- This is bad.
- We expect when the context is same, the behaviour exhibited by it should be consistent when approached
  directly and indirectly.
*/
```

- **08 dynamic binding**

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

class Base {
public:
	virtual void f() {
		cout << "From Base::f" << endl;
	}
};

class Derived : public Base {
public:
	void f() override {
		Base::f();
		cout << "From Derived::f" << endl;
	}
};

int main() {
	Derived derived;
	Base* pbase = &derived;
	pbase->f(); // Derived::f
	derived.f(); // Derived::f

}

/*
- Never override non virtual member functions of the base class.
- If overriding member function of the base class ensure that it is declared  virtual.
*/
```

- **09 pure virtual function**

```
class Shape {
public:
	virtual void Draw() const = 0;
};

class Triangle : public Shape {
public:
	void Draw() const override {}
};

class Rectangle : public Shape {
public:
	void Draw() const override {}
};

class Oval : public Shape { // 'Oval' has become an abstract class, because 'Shape::Draw' is not implemented in 'Oval'
};

void Draw(Shape* pshape) {
	pshape->Draw();
}

int main() {
	Shape shape; // Since 'Shape' is an abstract class, it can not be instantiated
	Shape* pshape = nullptr; // Pointer of 'Shape' (abstract class) can be defined

	Triangle t;
	Shape& rt = t; // Reference of 'Shape' (abstract class) can be defined
	Draw(&t);

	Rectangle r;
	Draw(&r);

	Oval o; // Since 'Shape::Draw' is not implemented in 'Oval', it has become an abstract class
	Draw(&o);
}

/*
- 'Draw' of 'Shape' is known as pure virtual function or abstract function.
- When we know certain operation exist in the base class but we don't know its
  implementation, then we declare such function as a pure virtual function or abstract function.
- Having one or more abstract functions in a class, makes that class an abstract class.
- Note 'Shape' class is now an abstract class.
- An abstract class cannot be instatiated.
```

- The class which can be instatiated is called as concrete class. For ex. 'Triangle', 'Rectangle' classes
  are concrete classes.
- If abstract function of a base class is not implemented in derived class then that derived class also
  becomes an abstract class. Oval is an example of derived class turned into abstract class.
- Pointer and reference of an abstract class can be defined.
- Abstract class besides abstract methods can have all sorts of members.
*/

- **10 polymorphism**

```
class Shape {
public:
	virtual void Draw() const = 0;
};

class Triangle : public Shape {
public:
	void Draw() const override {}
};

class Rectangle : public Shape {
public:
	void Draw() const override {}
};

class Oval : public Shape {
	void Draw() const override {}

};

void Draw(Shape* pshape) {
	pshape->Draw();
	//****This expression gives the polymorphic behaviour************
}

int main() {
	Triangle t;
	Draw(&t);

	Rectangle r;
	Draw(&r);

	Oval o;
	Draw(&o);

}
```

/*
- In the above code, pshape->Draw() draws the kind of graphical object with which pshape pointer is associated.
- 'pshape->Draw()' expression exhibits the polymorphic behaviour.
- Thus, an expression involving base pointer along with call to virtual function, exhibits polymorphic behaviour.
-***** The polymorphism exhibited by pshape->Draw() is known as runtime polymorphism.
-***** Function/operator overloading is known as compile time polymorphism.
*/

```
class Shape {
public:
	virtual void Draw() const = 0;
};

class Triangle : public Shape {
public:
	void Draw() const override {}
};

class Rectangle : public Shape {
public:
	void Draw() const override {}
};

class Oval : public Shape {
public:
	void Draw() const override {}

};

void Draw(Shape& shape) {
	shape.Draw();			//****This expression gives the polymorphic behaviour************
}

int main() {
	Triangle t;
	Draw(t);

	Rectangle r;
	Draw(r);

	Oval o;
	Draw(o);

}
```

/*
- A base class pointer and a base class reference

along with virtual function offers polymorphic behaviour
*/

```
class Shape {
public:
	virtual void Draw() const {};
};

class Triangle : public Shape {
public:
	void Draw() const override {}
};

class Rectangle : public Shape {
public:
	void Draw() const override {}
};

class Oval : public Shape {
public:
	void Draw() const override {}

};

void Draw(Shape shape) {
	shape.Draw();			//Note shape is no more a reference
	}
	// doesn't givw polymorphic behaviour.
```

As base class Shape is abstract class and we can not instantiate the abstract class instance

```
int main() {
	Triangle t;
	Draw(t);

	Rectangle r;
	Draw(r);

	Oval o;
	Draw(o);

}
```

- **11 object slicing**

Object slicing is a phenomenon in C++ that occurs when an object of a derived class is assigned to an object of a base class type. This can lead to the loss of the derived class's attributes and behaviors, as only the base class portion of the object is preserved. Essentially, when you assign a derived class object to a base class object, the extra data and functionality defined in the derived class are "sliced off."

```
class Base {
private:
	int m_i;
};

class Derived :public Base {
private:
	int m_j;
};

int main() {
	Base base1, base2;
	Derived derived1, derived2;

	base1 = base2;

	derived1 = derived2;

	base1 = derived2; //object slicing

	derived1 = base2;
}
```

- **12 vtable**

NEEDS TO BE PREPARE

- **13 interface**

```
class Iclonanable {
public:
	virtual void Clone() = 0; //pure virtual function
};

class ISerializable {
public:
	virtual void Serialize() = 0;
};

class Derived : public Iclonanable, public ISerializable {
public:
	void Clone() override{}
	void Serialize() override {}
};

int main() {
	//Iclonnable clonnable;
	Derived derived;
	derived.Clone();
	derived.Serialize();
}

/*
- An interface is an abstract class which contains public pure virtual functions
  and optionally virtual destructor.
```

```
- Thus every interface is an abstract class, however every abstract class is not an interface.
- In Microsoft programming, interface name is prefixed with letter 'I'.
- In CAA programming, interface name is prefixed with 'CATI'.
*/
```

- **14 upcasting downcasting**

```
class Base {};

class Derived : public Base {};

int main() {
	Derived derived;
	Base* pbase = &derived; // upcasting
}

/*
- Upcasting is converting derived class pointer or reference to base class pointer or reference resp.
- Upcasting happens implicitly.
*/
```

```
class Base {
public:
	virtual ~Base() {}
};

class Derived : public Base {};

class Derived2 : public Base {};

int main() {
	Derived derived;

	Base* pbase = &derived;

	Derived* pderived = nullptr;

	pderived = dynamic_cast<Derived*>(pbase); // This expression should work

	Derived2 derived2;

	pbase = &derived2;

	pderived = dynamic_cast<Derived*>(pbase); // his expression shouldn't work
}

/*
- Assigning base class pointer to derived class pointer is called as downcasting.
- Explicit casting is essential to perform downcasting.
- Use dynamic_cast operator to do downcasting.
- The base class must be polymorphic (should have presence of virtual function) for dynamic_cast to work.
- Adding virtual destructor to the base class would make base class polymorphic.
*/
```

```
#include <assert.h>
#include <typeinfo>

enum Color { Black, Red, Green, Blue, White };

class Shape {
public:
	virtual ~Shape() {}
public:
	virtual void Draw() = 0; // general member function
};

class Rectangle : public Shape {
public:
	void Draw() {}
	void Fill(Color color) {} // special member function
};

class Line : public Shape {
public:
	void Draw() {}
};

int main() {
	const size_t size = 2;
	Shape* pshape[size] = { new Rectangle, new Line };
	for (int i = 0; i < size; i++) {
		pshape[i]->Draw();
		if (typeid(*pshape[i]) == typeid(Rectangle)) {
			Rectangle* prect = dynamic_cast<Rectangle*>(pshape[i]); // Here because we want to call
			// special member function of Rectangle, we are doing downcasting
			assert(prect != nullptr);
			prect->Fill(Color::Blue);
		}
	}
	for (int i = 0; i < size; i++) {
		delete pshape[i];
		pshape[i] = nullptr;
```

```
	}
}
```

➢ 15 public vs protected vs private inheritance

```
class A {
public:
	int m_i;
protected:
	int m_j;
private:
	int m_k;
public:
	void f() {
		m_i = 1;
		m_j = 2;
		m_k = 3;
	}
};

class B : public A {
public:
	void g() {
		m_i = 1;
		m_j = 2;
		m_k = 3; // E
	}
};

class C : public B {
public:
	void h() {
		m_i = 1;
		m_j = 2;
		m_k = 3; // E
	}
};

int main() {
	A u;
	u.m_i = 1;
	u.m_j = 2; // E
	u.m_k = 3; // E

	B v;
	v.m_i = 1; // E
	v.m_j = 2; // E
	v.m_k = 3; // E

	C w;
	w.m_i = 1; // E
	w.m_j = 2; // E
	w.m_k = 3; // E
}
```

➢ 16. Copy constructor, overloaded assignment operator and inheritance

```
class Derived {};

/*
- List implicitly defined members of the Derived class
  - Because the class Derived is not written with any constructor, there
    exist compiler implemented default constructor. In future, if author of Derived class
    writes a constructor, compiler will stop supplying default constructor.
  - Other implicitly defined members are: copy constructor, copy assignment operator,
    move constructor, move assignment operator and destructor.
*/
```

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

class Base {
public:
	Base() {
		cout << "From Base Default Constructor" << endl;
	}

	Base(const Base& obj) {
		cout << "From Base Copy Constructor" << endl;
	}

	~Base() {
		cout << "From Base Destructor" << endl;
	}
	Base& operator=(const Base& obj) {
		if (this != &obj) { // checking for self assignment
			// memberwise assignment
		}
		return *this;
	}
};

class Derived : public Base {};

int main() {
	Derived u; // invokes compiler supplied default constructor of Derived class
	Derived v = u; // invokes compiler supplied copy constructor of Derived class
```

```
        v = u; // invokes compiler supplied copy assignment operator of Derived class
}

/*
- Compiler implemented
  - default constructor of derived class calls default constructor of base class.
  - copy constructor of derived class calls copy constructor of base class.
  - copy assignment operator of derived class calls copy assignment operator of base class.
  - destructor of Derived calls destructor of Base
*/
```

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

class Base {
public:
        Base() {
                cout << "From Base Default Constructor" << endl;
        }

        Base(const Base& obj) {
                cout << "From Base Copy Constructor" << endl;
        }

        ~Base() {
                cout << "From Base Destructor" << endl;
        }

        Base& operator=(const Base& obj) {
                if (this != &obj) { // checking for self assignment
                        // memberwise assignment
                }
                return *this;
        }
};

class Derived : public Base {
public:
        Derived() {
                cout << "From Derived Default Constructor" << endl;
        }

        Derived(const Derived& obj) : Base(obj) {
                cout << "From Derived Copy Constructor" << endl;
        }

        ~Derived() {
                cout << "From Derived Destructor" << endl;
        }

        Derived& operator=(const Derived& obj) {
                if (this != &obj) {
                        Base::operator=(obj);
                        // derived memberwise assignment
                }
                return *this;
        }
};

int main() {
        Derived u; // invokes compiler supplied default constructor of Derived class
        Derived v = u; // invokes compiler supplied copy constructor of Derived class
        v = u; // invokes compiler supplied copy assignment operator of Derived class
}


/*
- While implementing copy constructor and copy assignment operator
  in derived class, ensure that call is made to the copy constructor and
  copy assignment operator of the base class resp.
*/
```

➢ *DAY 15*

➢ 01 Virtual Destructor

```
#include<crtdbg.h>
class Base {
public:
        Base() {

        }
        ~Base() {

        }
};

class Derived : public Base {
public:
        Derived() {

        }
        ~Derived()  {
```

```
	}
};

int main() {
	Base* pobj = new Base();
	delete pobj;
	pobj = nullptr;
	_CrtDumpMemoryLeaks();
}

/*
- Expected is call to Base constructor upon creation of Base object and its happening.
- Expected is call to Base destructor upon deletion of Base object and its happening.
- Hence we conclude that above code is working OK.
*/
```

```
#include<crtdbg.h>
class Base {
public:
	Base() {

	}
	~Base() {

	}
};

class Derived : public Base {
public:
	Derived() {

	}
	~Derived() {

	}
};

int main() {
	Derived* pobj = new Derived();
	//Base(), Derived()
	delete pobj;
	//~Derived(), ~Base()
	pobj = nullptr;
	_CrtDumpMemoryLeaks();
}

/*
- Expected is call to Base constructor and then Derived constructor
  upon creation of Derived object and its happening.
- Expected is call to Derived destructor and then Base destructor
  upon deletion of Derived object and its happening.
- Hence we conclude that above code is working OK.
*
```

```
#include<crtdbg.h>
class Base {
public:
	Base() {

	}
	virtual ~Base() {

	}
};

class Derived : public Base {
public:
	Derived() {

	}
	~Derived() {

	}
};

int main() {
	Base* pobj = new Derived();
	//Base(), Derived()
	delete pobj;
	//~Derived(), ~Base()
	pobj = nullptr;
	_CrtDumpMemoryLeaks();
}

/*
- It is strongly recommended to write virtual destructor in base class even if it is empty.
- Though base class may not consume resource, derived class may consume.
*/
```

- **02 Mutliple class Inheritance**

```
class Base1 {
public:
	void f() {}
};

class Base2 {
public:
	void g() {}
};

class Derived : public Base1, public Base2 {

};
```

```cpp
int main() {
	Derived d;
	d.f();
	d.g();
}

/*
- Above is an example of multiple inheritance.
- Implementation of Base1 and Base2 are inherited in Derived class.
- Hence above inheritance is also known as implmentation inheritance.
- Implementation inheritance is also known as class inheritance.
*/
```

- **03 multiple interface inheritance**

```cpp
class IBase1 {
public:
	virtual void f() = 0;
};

class IBase2 {
public:
	virtual void g() = 0;
};

class Derived : public IBase1, public IBase2 {
public:
	void f() override {}
	void g() override {}

};

int main() {
	Derived d;
	d.f();
	d.g();
}

/*
- Above is an example of multiple interface inheritance.
*/
```

- **04 Diamond Problem**

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

class SuperBase {
public:
	void Print() {
		cout << m_a << endl;
	}
private:
	int m_a;
};

class Base1 : public SuperBase {
private:
	int m_b;
};

class Base2 : public SuperBase {
private:
	int m_c;
};

class Derived : public Base1, public Base2 {
private:
	int m_d;
};

int main() {
	Derived d;
	d.Print();
}
```

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

class SuperBase {
public:
	SuperBase(int a) : m_a(a) {}
public:
	void Print() {
		cout << m_a << endl;
	}
private:
	int m_a;
};

class Base1 : virtual public SuperBase {
public:
	Base1(int a, int b) : SuperBase(a), m_b(b) {}
private:
	int m_b;
};
```

```
class Base2 :virtual public SuperBase {
public:
	Base2(int a, int c) : SuperBase(a), m_c(c) {}
private:
	int m_c;
};

class Derived : public Base1, public Base2 {
public:
	Derived(int a1, int b, int a2, int c, int d, int a3)
		: Base1(a1, b), Base2(a2, c), m_d(d) {}
private:
	int m_d;
};

int main() {
	Derived d(1, 2, 3, 4, 5,6);
	//d.Print();
}
```

➢ DAY 16

➢ 01 smart pointer

```
#include <crtdbg.h>

class Dummy {
public:
	void F();
};

void Dummy::F() {}

void G() {
	Dummy* px = new Dummy;
	px->F();
}

int main() {
	G();
	G();
	_CrtDumpMemoryLeaks();
}

/*
- Everytime 'G' is called, memory is leaked.
*/
```

```
#include <crtdbg.h>
class Dummy {
public:
	void F();
};

void Dummy::F() {}

class DummySMP {
public:
	DummySMP(Dummy* pobj);
	~DummySMP();
public:
	Dummy* GetObject() {
		return m_pobj;
	}
private:
	Dummy* m_pobj;
};

DummySMP:: DummySMP(Dummy* pobj) :
m_pobj(pobj){}

DummySMP:: ~DummySMP() {
	delete m_pobj;
	m_pobj = nullptr;
}

void G() {
	DummySMP dummySMP(new Dummy);
	Dummy* pobj = dummySMP.GetObject();
	pobj->F();
}

int main() {
	G();
	G();
	_CrtDumpMemoryLeaks();
}
```

```
#include <crtdbg.h>
class Dummy {
public:
	void F();
};

void Dummy::F() {}

class DummySMP {
public:
	DummySMP(Dummy* pobj);
	~DummySMP();
public:
	Dummy* operator->() {
		return m_pobj;
```

```
	}
private:
	Dummy* m_pobj;
};


DummySMP::DummySMP(Dummy* pobj) :
m_pobj(pobj) {}

DummySMP:: ~DummySMP() {
	delete m_pobj;
	m_pobj = nullptr;
}

void G() {
	DummySMP dummySMP(new Dummy);
	dummySMP.operator->()->F();  // Cascading
}

int main() {
	G();
	G();
	_CrtDumpMemoryLeaks();
}
```

```
#include <crtdbg.h>
class Dummy {
public:
	void F();
};

void Dummy::F() {}

class DummySMP {
public:
	DummySMP(Dummy* pobj);
	~DummySMP();
public:
	Dummy* operator->() {
		return m_pobj;
	}
private:
	Dummy* m_pobj;
};


DummySMP::DummySMP(Dummy* pobj) :
m_pobj(pobj) {}

DummySMP:: ~DummySMP() {
	delete m_pobj;
	m_pobj = nullptr;
}
```

```
void G() {
	DummySMP dummySMP(new Dummy);
	//"dummySMP" is a smart pointer
	dummySMP->F();			//dummySMP->F(); is executed as good as dummySMP.operator ->()->F();
}

int main() {
	G();
	G();
	_CrtDumpMemoryLeaks();
}

/*
- Smart pointer essentially is an object.
- It takes responsibility to manage lifetime of dynamically allocated object of another class.
- It pretends to be like a pointer but actually it is not a pointer.
*/
```

```
#include <memory>
#include <crtdbg.h>
using namespace std;

class Dummy {
public:
	void F();
};

void Dummy::F() {}

void G() {
	unique_ptr<Dummy> u(new Dummy);
	u->F();

	//unique_ptr<Dummy> v = u; // Error: unique_ptr owned object cannot be owned by two or more unique_ptr objects

	//unique_ptr<Dummy> w;
	//w = u; // Error: unique_ptr owned object cannot be owned by two or more unique_ptr objects

	unique_ptr<Dummy> x;
	x.swap(u); // Its possible to transfer ownership from one unique_ptr object to another unique_ptr object
}

int main() {
	G();
```

```
	G();
	_CrtDumpMemoryLeaks();
}

/*
- Introduction to unique_ptr.
- unique_ptr object claims exclusive ownership over the dynamically allocated object
  given to it for its life management.
*/
```

```
#include <memory>
#include <crtdbg.h>
using namespace std;

class Dummy {
public:
	void F();
};

void Dummy::F() {}

void G() {
	shared_ptr<Dummy> u(new Dummy);
	u->F();

	shared_ptr<Dummy> v = u; // OK: 'u' can share ownership with 'v'.

	shared_ptr<Dummy> w;
	w = u;

	shared_ptr<Dummy> x;
	x.swap(u); // Its possible to transfer ownership from one shared_ptr object to another shared_ptr object
	                         // Now w and v will be the two owners of the object

}

int main() {
	G();
	G();
	_CrtDumpMemoryLeaks();
}

/*
- shared_ptr owned object can be owned by two or more shared_ptr objects.
*/
```

- **03 exception handling**

```
#include <iostream>

class Circle {
private:
	int m_radius;
public:
	void SetRadius(int radius) {
		if (radius < 0)
			throw 101;
		m_radius = radius;
	}
};

int main() {
	try {
		Circle a;
		a.SetRadius(-5);
	}
	catch (int e) {
		std::cout << e << std::endl;
	}
}
```

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

class Exception {
public:
	Exception(int errorcode, const char* description, const char* pfunctionname, int lineno)
		: m_errorcode(errorcode), m_lineno(lineno) {
		strcpy_s(m_description, 64, description);
		strcpy_s(m_functionname, 64, pfunctionname);
	}
public:
	int GetErrorCode() const { return m_errorcode; }
	const char* GetDescription() const { return m_description; }
	const char* GetFunctionName() const { return m_functionname; }
	int GetLineNo() const { return m_lineno; }
private:
	int m_errorcode;
	char m_functionname[64];
	char m_description[64];
	int m_lineno;
};
```

```
class Circle {
private:
	int m_radius;
public:
	Circle(int radius) {
		SetRadius(radius);
	}
public:
	void SetRadius(int radius) {
		if (radius < 0)
			throw Exception(1001, "Radius 
must be +ve number.", __FUNCTION__, __LINE__);
		m_radius = radius;
	}
};

int main() {
	try {
		Circle a(10);
		a.SetRadius(-5);
	}
	catch (Exception& e) {
		cout << e.GetErrorCode() << endl;
		cout << e.GetDescription() << endl;
		cout << e.GetFunctionName() << endl;
		cout << e.GetLineNo() << endl;
	}
}

/*
A try block can have multiple catch blocks but a catch 
block cannot have multiple try blocks.
The exception handling mechanism checks for the first 
catch block that can refer to type of exception 
object thrown. As soon as if finds same, that catch block 
is executed and no other catch block is checked.
Nesting of try...catch is possible.
One can throw exception from constructor but should 
not be thrown from destructor.
While writing multiple catch blocks, arrange catch 
blocks from special exception types to 
general exception type.
Once the exception object escapes main function, it 
breaks the code.
One can rethrow exception by writing just "throw;" in 
catch block.
*/
```

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

class Exception {
public:
	Exception(int errorcode, const char* 
description, const char*  pfunctionname, int lineno)
		: m_errorcode(errorcode), 
m_lineno(lineno) {
		strcpy_s(m_description, 64, 
description);
		strcpy_s(m_functionname, 64, 
pfunctionname);
	}
public:
	int GetErrorCode() const { return m_errorcode; 
}
	const char* GetDescription() const { return 
m_description; }
	const char* GetFunctionName() const { return 
m_functionname; }
	int GetLineNo() const { return m_lineno; }
private:
	int m_errorcode;
	char m_functionname[64];
	char m_description[64];
	int m_lineno;
};

class InvalidRadiusException : public Exception {
public:
	InvalidRadiusException(const char* 
pfunctionname, int lineno)
		: Exception(101, "Radius must be +ve 
number.", pfunctionname, lineno) {}
};

class Circle {
private:
	int m_radius;
public:
	Circle(int radius) {
		SetRadius(radius);
	}
public:
	void SetRadius(int radius) {
		if (radius < 0)
			throw 
InvalidRadiusException(__FUNCTION__, __LINE__);
		m_radius = radius;
	}
};

int main() {
	try {
		Circle a(10);
		a.SetRadius(-5);
	}
	catch (InvalidRadiusException& e) {
		// If we want to handle 
InvalidRadiusException in some different way then
```

```
            // we need to implement its dedicated
catch block.
            cout << e.GetErrorCode() << endl;
            cout << e.GetDescription() << endl;
            cout << e.GetFunctionName() << endl;
            cout << e.GetLineNo() << endl;
            //throw; // rethrows the exception
        }
        catch (Exception& e) {
            cout << e.GetErrorCode() << endl;
            cout << e.GetDescription() << endl;
            cout << e.GetFunctionName() << endl;
            cout << e.GetLineNo() << endl;
        }

}
```

- **06 enumeration**

```
enum class Colour {
        Black,
        Red,
        Green,
        Blue=2,
        White
};

class Rectangle {
public:
        void Fill(Colour Colorcode) {

        }
};

int main() {
        Rectangle r;
        r.Fill(Colour::White);
        r.Fill(1); //Note int cannot be passed as an
argument
}
/*
- The underneath data type of enum is int.
- Specific value can be assigned to enum constants.
- When no value is assigned to first enum constant, it
start with 0.
- The value of enum constant is calculated as 1 + value
of previous enum constant.
- Function parameter can be enum type.
- In such case, only enum permitted constants can be
passed as argument to it.
- Though int is underlying data type of enum, it cannot
be passed as an argument.
*/
```

- **07 friend function**

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

class Integer {
public:
        Integer(int i = 0);
private:
        int m_i;
        friend int main();
};

Integer::Integer(int i) : m_i(i) {}

int main() {
        Integer u(10);
        cout << u.m_i << endl;
}

/*
- Friend function gets privilege to access private
members directly.
- Donot use friend functions.
*/
```