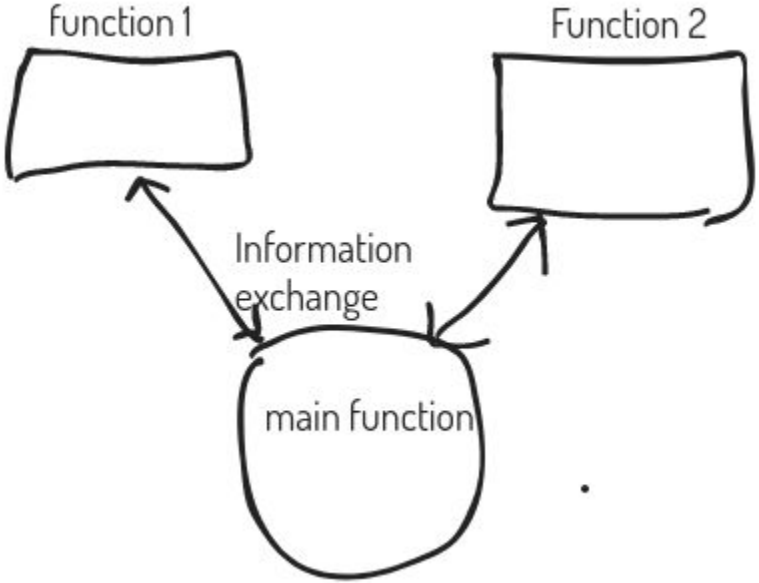
Module 1 : Introduction to Object Oriented Concepts

A Review of structures, Procedure–Oriented Programming system, Object Oriented Programming System, Comparison of Object Oriented Language with C, Console I/O, variables and reference variables, Function Prototyping, Function Overloading. Class and Objects: Introduction, member functions and data, objects and functions.

Structure vs Object oriented programming

C++ language supports both structure oriented and object oriented programming.

Program is made up of 2 entities → variables and statements/functions



Structure vs Object oriented programming

In structure-oriented programming importance is given to functions rather than variables/data.

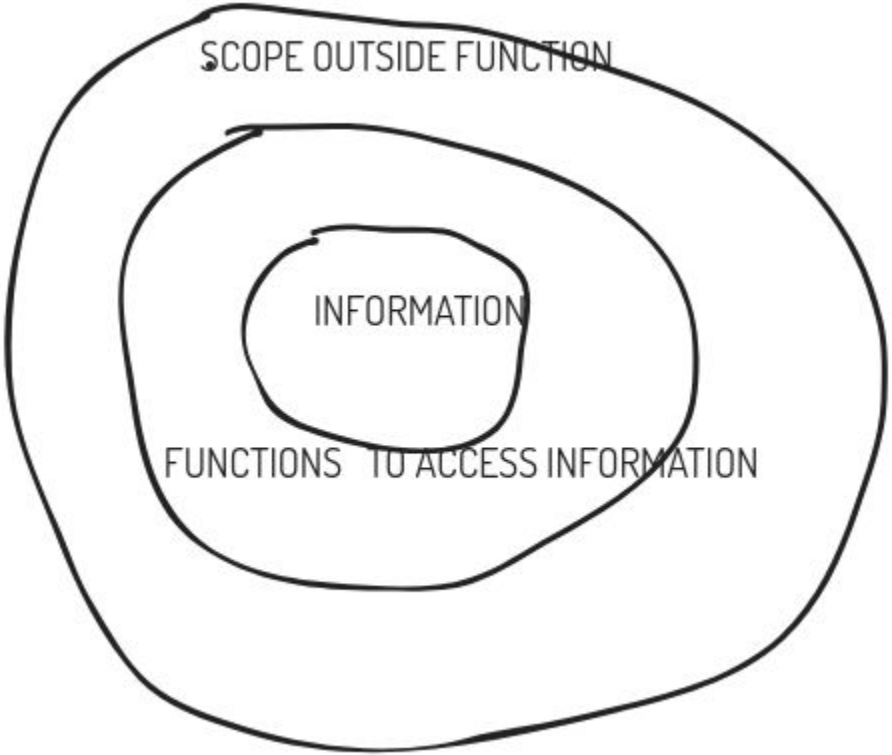
More importance will be given to split the complex program into modules/functions and less importance is given to movement of data.

In **OOP (Object Oriented Programming)** more significance is given to information/data.

Main aim of OOP is to secure the data.

Importance is given to information first and around it functions are created to access the same.

Structure vs Object oriented programming

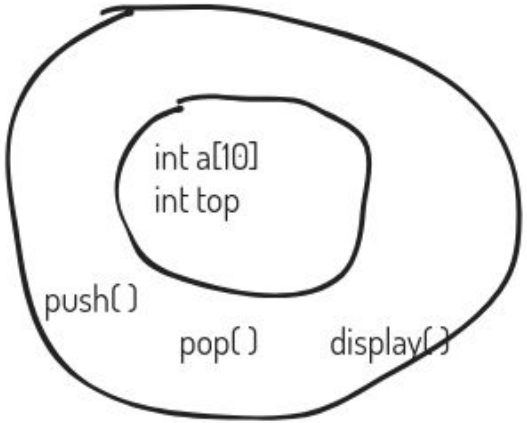


Structure vs Object oriented programming

Ex: consider the simulation of a stack program,
variables without which stack cannot be simulated are
array (int a[10]) top pointer (int top)

These two variables are important from stack program perspective and they have to be secured in OOP.

The functions **push( ), pop( )** and **display ( )** will surround these variables.



C++ language

C++ language supports both structure oriented and object oriented programming approach.

All C++ program source files must have .cpp extension
Ex: hello.cpp

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

int   main( )
{
      cout<<" hello c++";

}
```

cout is a variable/object of type **ostream** structure/class which is built-in

<< is termed as **INSERTION** operator

C++ language - Reading value from standard input device

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

int main( ){

    int   i;
    cout<<" Enter value for i"<<endl;
//endl will place the cursor to next line after printing the contents in cout statement
//endl effect is similar to '\n'.

    cin>>i;          cout<<"Value of i is"<<i;  }
```

cin is a variable/object of type **istream** built-in data-type/class-type

>> is termed as **EXTRACTION** operator

C++ language - Access types in C++

Information in C++ can be access using 3 ways
1. Using name of the variable (named access)
2. Using address of the variable (address access)
3. Using alias of the variable (reference access)

Ex:

```
int i=10;   int *p=&i;
cout<<i<<endl;
cout<<(*p)<<endl;
```

Reference variables

Reference variables are aliases for already existing variables.

G.F: **<data-type> &<alias-name> = <already-existing-variable-name>;**

Ex:
```
int i=10;
int &r = i;
```

C++ language - Reference variables

Ex:
```
int i=10;
int &r = i;
```

‘r’ is to be considered as an another name for the memory, which is already named as ‘i’.

In other words, ‘r’ and ‘i’ are two different names for the same memory.

Ex:
```
cout<<"Address of  i  is "<< &i;
cout<<"Address of  r  is "<< &r;
```

If the above two statements prints the same address then it is proved that ‘r’ is an alias name for already existing variable ‘i’.

C++ language - Reference variables

Rules to create and use Reference variables.

1. Reference variables must be initialized at the point of creation.
   Ex: int &r; // CTE

2. Once an alias is created modification done to memory contents via alias name will be reflected on actual name and vice-versa.
   Ex: int i=10; int &r=i;
   r++
   cout<<i;

3. Once reference variables are bound to a variable, it cannot be made to refer to any other variable even of the same type.
   Ex: int i=10, j=20;
   int &r=i;
   &r=j; // CTE

C++ language - Reference variables

Rules to create and use Reference variables.

4.Multiple references can be created for the same variable.

Once alias or reference is created it can be used to create another alias

Ex: int i=10; int &r = i; int &s=i; int &t=r;

There is no much usage of using the alias name or pointer in the same scope of the variable.

Ex: int i=10;
int *p=&i;
int &r=i;

C++ language - Parameter passing techniques
1. Call by value
2. Call by address
3. Call by reference

Call by reference
Formal parameters in function definition will have an alias
Actual parameters in function call will have actual variables.

Ex:

```
void accept(int &a, int &b)  // formal parameters a and b
{  cin<<a;   cin<<b;  }


int m,n;
accept(m,n);  // actual parameters m and n
```

Call by reference or Call by address must be used,
when the modification done in one scope must be seen across the scope on variables.

C++ program to swap two complex numbers using function

C++ language - Namespace

Namespace in C++ provides the facility to name a scope.

C++ language provides enormous built-in facility via built-in functions and classes(similar to structures).
Ex: information can be stored in a list data structure form, which is built-in to the language

Hence, enormous amount of names has been used within the language itself.
Ex: if programmer of c++ wants to create a function swap, that name is already present in the language facility.

Hence, To avoid name clashes, **namespaces** are used.

```
Ex:  namespace stack                    namespace queue
     {                                  {
        int   i;                            int i;
     }                                  }  // ; not required
```

Two different variables with same name can be declared inside a namespace, the namespace name avoids the name clash.

“std” is the built-in namespace in C++ language.

C++ language - Namespace

Inside namespace program segments like

Variable declarations
Function definitions
class/structure declarations
Constant declarations etc., can be done except main function

Identifiers that are inside namespaces can be used in any function/program using two ways

1. By “using namespace <namespace-name>” keyword

2. By “<namespace-name> :: <identifier-inside-namespace>” scope resolution operator

1. By “using namespace <namespace-name>” keyword

At any point in the program using keyword can be used and all the identifiers in that namespace can be used automatically.

C++ language - Namespace

Ex:

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

namespace stack {
    int i;
    struct comp
    {
      int r,i;
    };
}


int main()  {
        using namespace stack;
          i=9;       cout<<i;

          comp a;

          a.r=690;     cout<<a.r;
        return 0;
}
```

C++ language - Namespace

The identifiers inside the namespace stack can be used only inside the main scope, and not beyond that.

Ex:

```
int main( ) {
        using namespace stack;
        ….
}

void accept( ) {
        i=90;   //generates CTE
}
```

If the identifiers inside the namespace stack has to be used in a file, then the “using…” line must be coded above the main function.

Ex:

```
using namespace stack;
int main( ) {
                ….
}

void accept( ) {
        i=90;   // no error
}
```

C++ language - Namespace
2. By "<namespace-name> :: <identifier-inside-namespace>" scope resolution operator

Identifiers inside a namespace can also be accessed using namespace name and :: (Scope resolution operator), without typing using keyword.
Ex:

```
stack::i=90;
cout<<stack::i;
```

If :: is used, then each time whenever an identifier is referred to within the namespace, the namespace name with :: has to be used.

3 main characteristics of OOP

1. Encapsulation
2. Polymorphism
3. Inheritance

Encapsulation: combining **data** with **functions** is termed as encapsulation

Polymorphism: poly=many morph=forms. a single function name can be used repeatedly in the same program for several purposes. Function overloading Function overriding

Inheritance: reusing the code and data present in another class.

C++ language - Function overloading

Function overloading is the concept used under Polymorphism characteristics of OOP.

Same function name can be used in several functions provided the task is same but the task is applied on different data types (and on different parameter set).

Ex: addition of numbers
type of numbers can be int - 2 numbers
type of numbers can be int - 3 numbers
type of numbers can be float - 2 numbers
type of numbers can be unsigned it - 3 numbers

Function overloading is defined as follows
Return type may or may not be same.
Function name must be same
Parameters must differ either in the number or in the type.

Ex: void add(int ,int);
void add(int, int,int);
these two are differing in the number of parameters (type is same)

void diff(int, int);
void diff(float, float);
these two are differing in the type of parameters (no of parameters are same)

C++ language - Function overloading

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

void add(int a, int b)
{  cout<<"+ of 2 int nos "<<(a+b)<<endl;  }


void add(int a, int b,int c)
{   cout<<"+ of e int nos "<<(a+b)<<endl;  }


void add(float a, float b)
{  cout<<"+ of 2 float nos "<<(a+b)<<endl;   }

int main(){

    add(1,2);
    add(1,2,3);
    add((float)1.1,(float)2.1);  }
```

Type casting of value is done because by default 1.1 is treated as a double value by compiler and it will search for a function with formal parameter type double.

C++ language - Default values for Formal parameters

In C++ the formal parameters can be initialized with a default value.

```
Ex:   void  total_bill(int unit_price, int quantity=1)
      {
        …
      }
```

If a formal parameter is initialized then it must be coded to the right most side of formal parameter list.

If a formal parameter is initialized then that argument can receive any one value, either the initialized value or the passed actual parameter value.

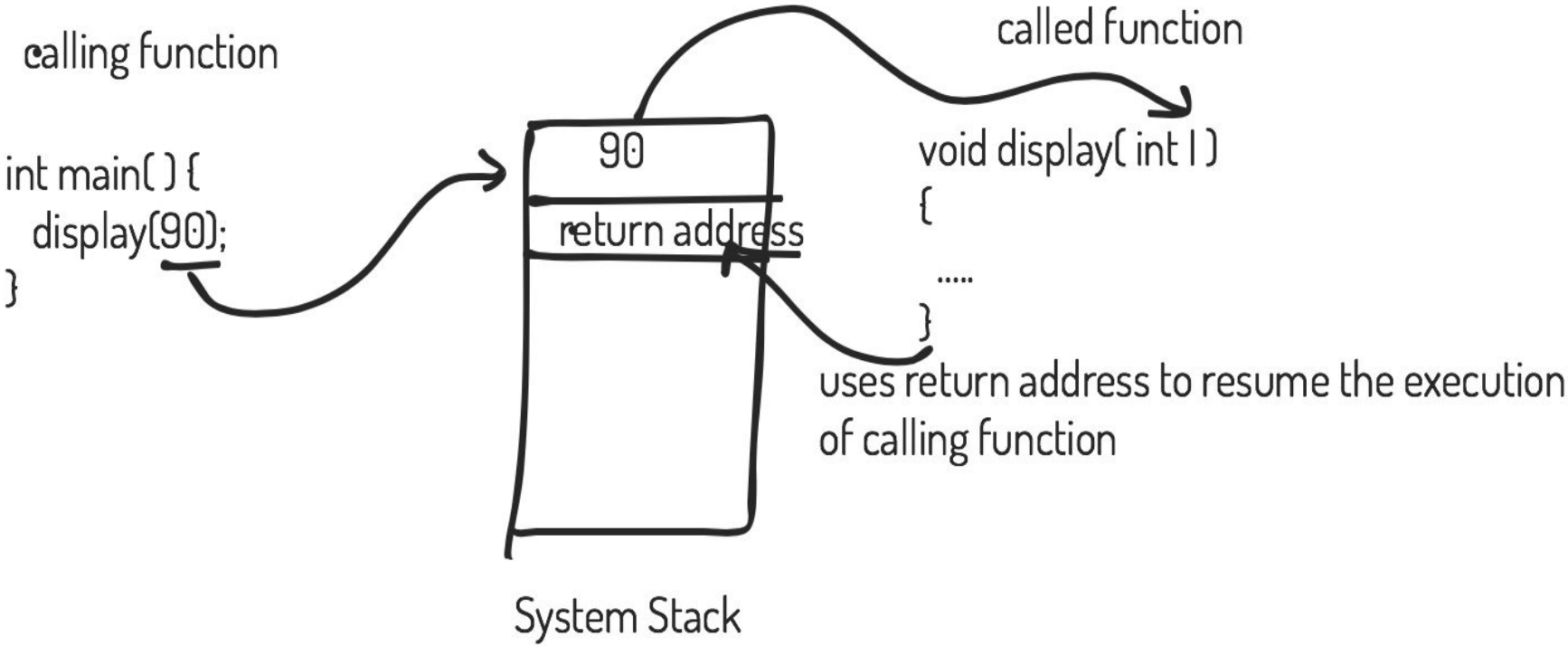
Ex:

```
void  total_bill(int, int = 1);       // void total_bill(int=1, int); generates CTE
int   main( ) {
            total_bill(100);
            total_bill(100, 2);
}
void total_bill(int unit_price, int quantity)
{
      cout<<quantity;
}
```

1st output will 1 the default value. 2nd output will be 2 the passed value from function call.

C++ language - Inline function

Execution of a normal function consumes more CPU cycles and hence accounts for more execution time.



C++ language - Inline function

In order to reduce the execution time of a program, C++ provides an another form of function termed as inline function(macro).

Inline function reduces the execution time of a program, by replacing the statements before compilation itself.

Inline function has to be coded by placing the inline keyword as the first word in the function definition.
Ex: inline void display(int i)
{ …}

Only the function which contain less number of statements must be coded as inline function in C++.

If function contains a recursive call or a looping construct then inline keyword must not be used, if used compiler neglects the request and the function will be executed as a normal function.
Ex:

```
inline void display(int);
int   main( ) {
            display(10);
}
void display(int i)
{  cout<<i;  }
```

C++ language - Encapsulation

Combining both data and functions together in a scope.

Import data/variables will be given significance rather than all the variables/data in the program.

Functions are the one which represent the logical modification that must be done on data.

Ex: program to simulate stack activity
Important variables are: int a[20], int top;
Important functions are: push(), pop(), display();

```
struct stack {
      int a[20],top;
      void push();  void pop();    void display();
};
```

In C++ variables and functions can be combined together termed as encapsulation

Functions push( ), pop( ) and display( ) are in structure scope and can access array ‘a’ and ‘top’.

A variable of type stack can be used to access array and top and also can invoke functions push….

C++ language - Encapsulation

Program to simulate queue activity
Important variables are: int a[20], r, f;
Important functions are: delete_front(), insert_rear(), display();

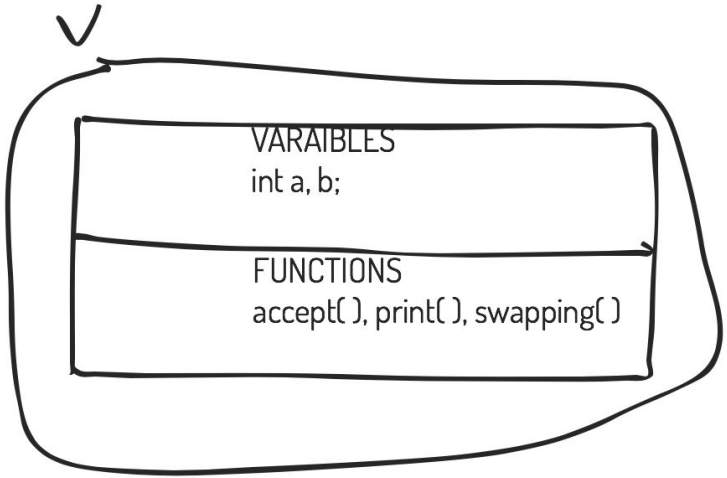
```
struct queue {
        int a[20], r, f;
        void insert_rear();
        void delete_front();   void display();
};
```

Program to simulate swapping of two numbers using encapsulation
Important variables are: int a,b;
Important functions are: accept( ), print( ), swapping( );

```
struct swap {
        int a,b;
        void  accept( );
        void  print( );
        void  swapping( );
};
```

C++ language - Encapsulation
Ex: swap expansion

```
struct swap {
        int a,b;

        void  accept( )
        {  cin>>a>>b;  }

        void  print( )
        { cout<<a<<b; }

        void  swapping( )
        {  int t=a;
           a=b;
           b=t;
        }
};//end of structure

int main( )
{
        swap  v;    cout<<sizeof(v); }
```



C++ language - Encapsulation

Since, structure in C++ contains not only variables but also functions, functions can also be called on variables of type structure

v.accept( );

Variable ‘v’ contains a and b and function accept accepts the value for a and b which is in v.

Similarly, if display( ) function is called via v the contents of a and b stored in v will be displayed.

Also, if swapping( ) function is called via v the contents of a and b will be swapped.

Ex:

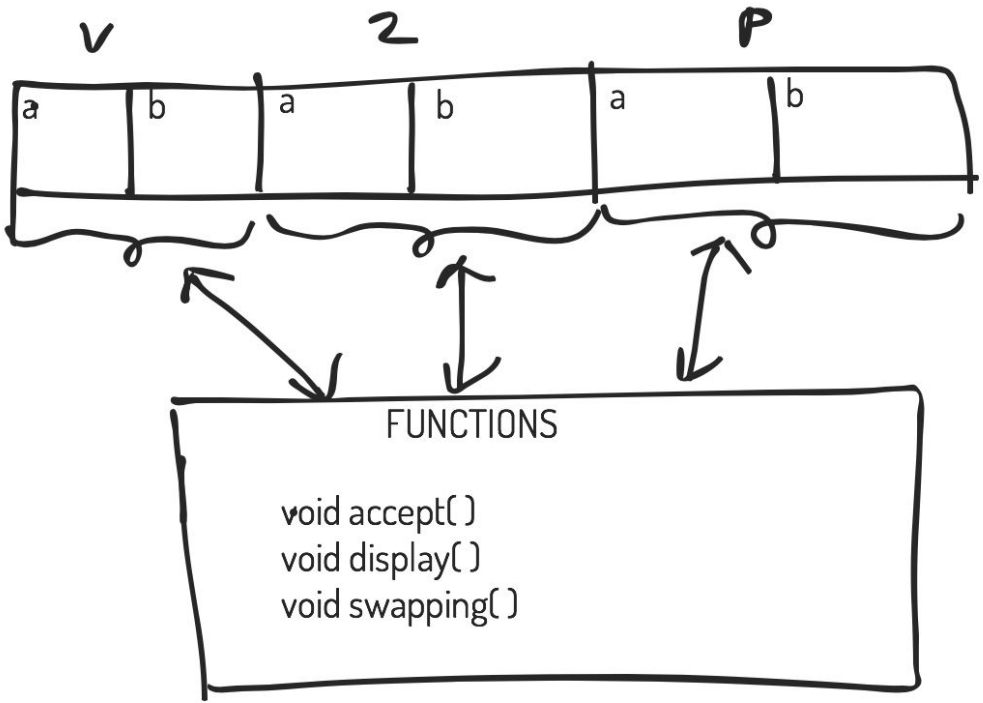
```
int main( ) {
        swap v;
        v.accept( );
        v.swapping( );
        v. display( );
}
```

If another variable of type swap is created say swap z; then z contains its own set of variables.

But only one set of functions will be maintained for both v and z.

C++ language - Encapsulation

Irrespective of how many variables of type swap is created in C++ program only one set of functions will be used on all these variables.



C++ language - Encapsulation

Variables inside struct are termed as DATA MEMBERS
Functions inside struct are termed as MEMBER FUNCTIONS in C++.

Main aim of OOP is to provide **DATA SECURITY.**

To achieve this variables inside struct need to be confined within struct and must not be allowed to access outside the struct.

In order to do this 3 keywords are used in C++ termed as **ACCESS SPECIFIERS**.
1. private
2. public
3. protected

If any data member or member function is used under private access specifier, it's access will be limited within the struct scope and cannot be accessed outside the struct scope.

In order to achieve data security data members inside struct must be declared under private access specifier.

```
struct   swap {
        private:   int a,b;
        ….
        ….
};
```

C++ language - Encapsulation

```
struct   swap {
        private:   int a,b;
        ….
        ….
};
```

data members a and b are now accessible within struct swap and from not outside of it.

Member functions can be declared under public access specifier, because even if they are declared under private, nothing can be accessible outside the struct scope and declaration of struct scope itself is not required.

```
struct   swap {
        private:   int a,b;
        public:
                        void accept( ){ …}
                        void display(){....}
                        void swapping( ) { …. }
};
```

C++ language - Encapsulation

Class

class is a keyword in C++ which provides the same facility as a struct keyword in C++.

class creates a new data type in C++.

class is used to exhibit encapsulation property of OOP.

Ex:
```
class  swap {
        private:   int a,b;
        public:
                    void accept( ){ …}
                    void display(){....}
                    void swapping( ) { …. }
};
```

The only difference between struct and class in C++ is

By default members in struct are public

By default members in class are private (The word default confines to not using any access specifiers)

C++ language - Encapsulation

Class

Ex:

```
struct swap {
        int i;
};

swap a;
a.i=10;
```

```
class swap1{
        int i;
};

swap1   a;
a.i=20; // CTE
```

Variables of type struct or class in C++ are also termed as **OBJECTS.**

Object/s exhibit encapsulation because these are variables of type class template.

Object is also considered as an **INSTANCE** (meaning occurrence) of class.

As memory is not allocated for a struct declaration, even class declaration will not get any memory allocated and hence termed as a **TEMPLATE.**

When a variable of struct is created memory is allocated to variables defined inside struct.

Similarly, **when an object is created** (of type class) memory is allocated for all the data members inside the class.

C++ language - Encapsulation

Class

As pointers, reference variables and arrays can be created for a struct type, similarly pointers and reference variables and arrays of class type can also be created.

Ex:

```
class student {
        private:   char na[20], usn[20];
        public:
                        void accept( ) { cin>>na>>usn; }

                        void display( ) { cout<<na<<" "<<usn<<endl; }
};

int  main( )   {
                student  p;   p.accept( );   p.display( );

                student  *q;  q=&p;    q->display( );

                student &r=p;    r.display( );

}
```

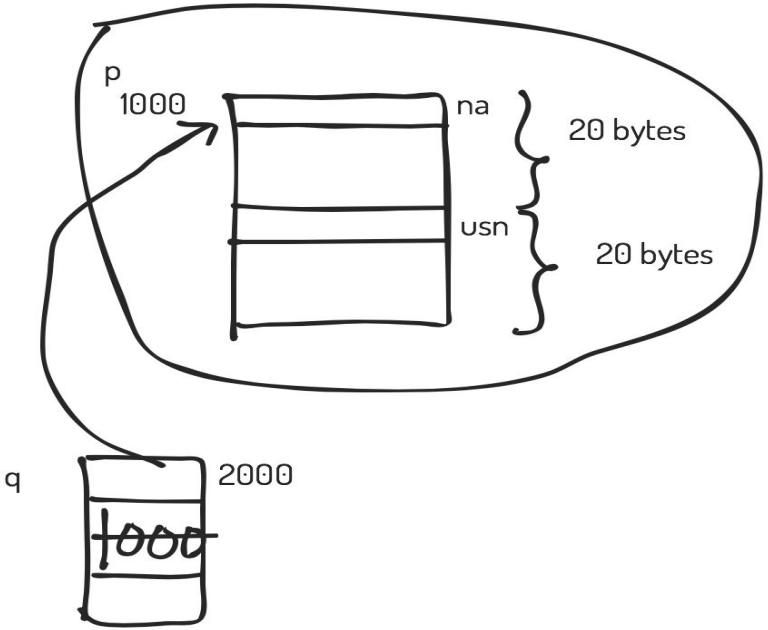
C++ language - Encapsulation - Class

Consider **student *q; - pointer of class student type**

q is a pointer of class type and acquires 8 bytes of memory.

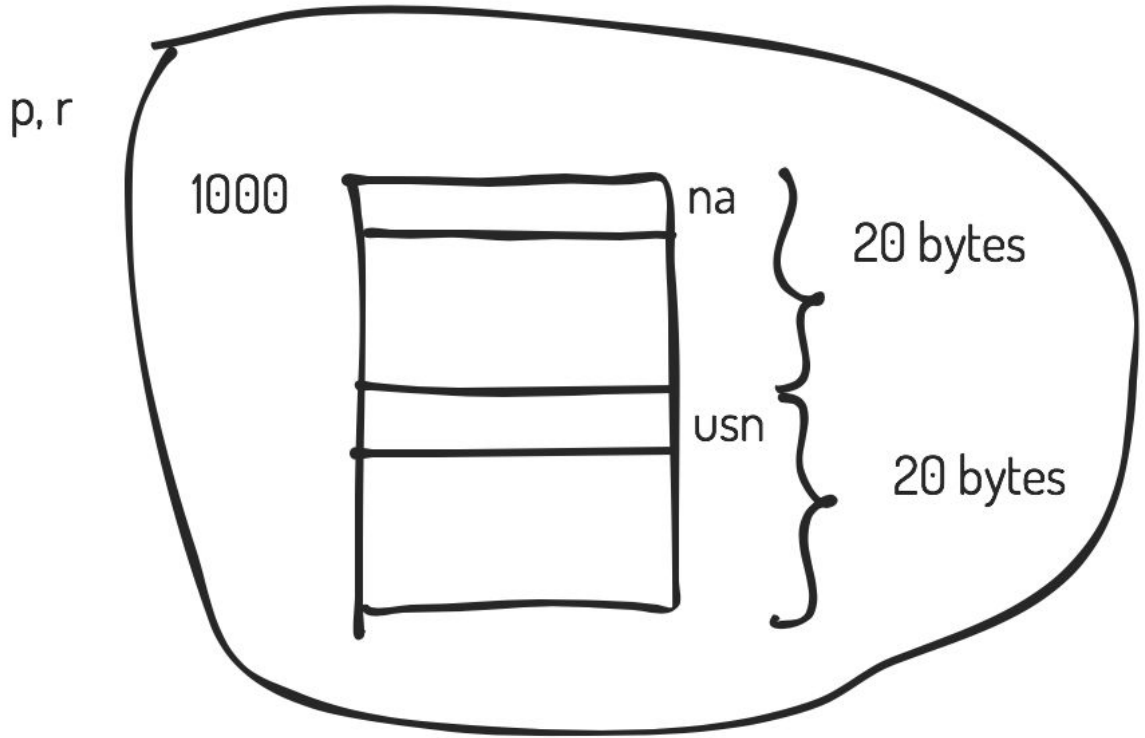
Without assigning a valid address for q the members in student (data members & member functions) cannot be accessed, if done executor generates segmentation fault, hence **q=&p** is essential.

"->" operator must be used to access members of a class, when an address of class type is used.



C++ language - Encapsulation - Class
Consider **student &r=p;**

'r' is an alias of p. r & p are different names for the same memory location.



C++ language - Encapsulation - Class

Ex:

```
class student {
        private:   char na[20], usn[20];
        public:
                        void accept( ) { cin>>na>>usn; }

                        void display( ) { cout<<na<<" "<<usn<<endl; }
};

void  main( )   {
                student  p[3];   // Array of class student type
// p represents the base address of the array
//p[0] is a student type object made up of name and usn
//p[1] is a student type object made up of name and usn …

                for(int i=0;i<4;i++)
                        p[i].accept();   // 4 set of name and usn has to be entered

                for(int i=0;i<4;i++)
                        p[i].display();  // 4 set of name and usn will be displayed
}
```

[ ]

Exp1 [exp2] ⇒ *(exp1 + exp2)

Either exp1/exp2 must yield an address and the other expression must yield an integer value.

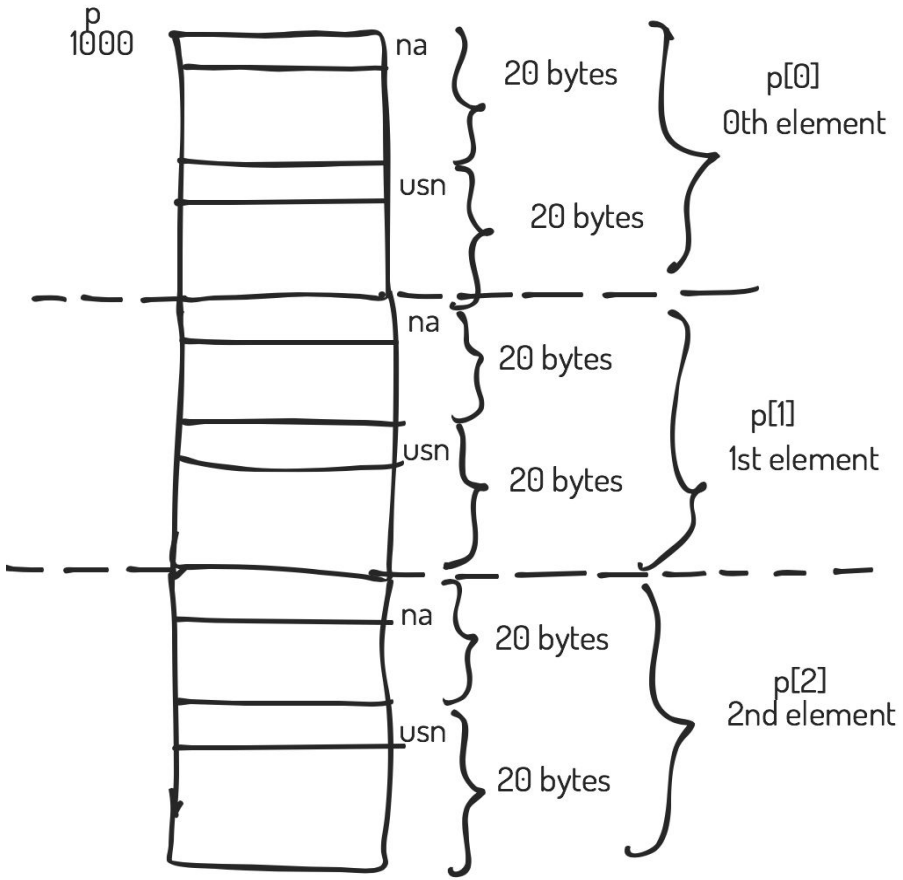
int a[2];

a[0]=89;

a[0] ⇒ *(a+0)

C++ language - Encapsulation - Class

Consider **student p[3];**



C++ language - Dynamic memory allocation

Dynamic memory allocation is the facility to allocate memory to store information during execution of the program. G.F <pointer-name> = new <data-type>;

Keywords used in C++ language to allocate memory is “new”.

To release or deallocate dynamically acquired memory keyword used is “delete”.

Dynamically acquired memory using new returns an address and a pointer must be used to store the same.

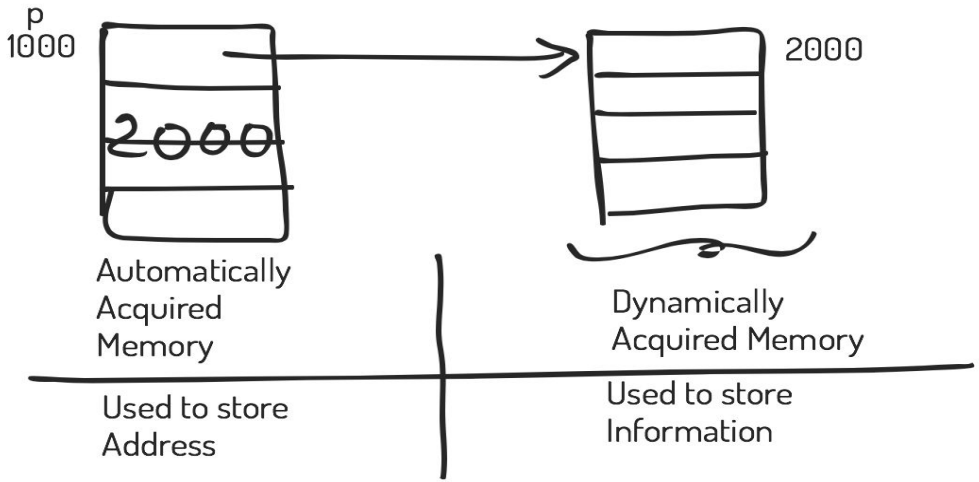
Ex: to allocate dynamic memory to store an information of type int.

```
int  *p;

p = new int;

cin>>(*p);
```

Because address cannot be passed to cin

```
cout<<*p;

delete p;
```

Memory addressed 2000 is released for further usage.



C++ language - Dynamic memory allocation

Dynamically acquired memory can be duly initialized with a value.

```
Ex:   int *p = new int(89);
      cout<<*p;
```

Value of 89 will be stored in the dynamically acquired memory.

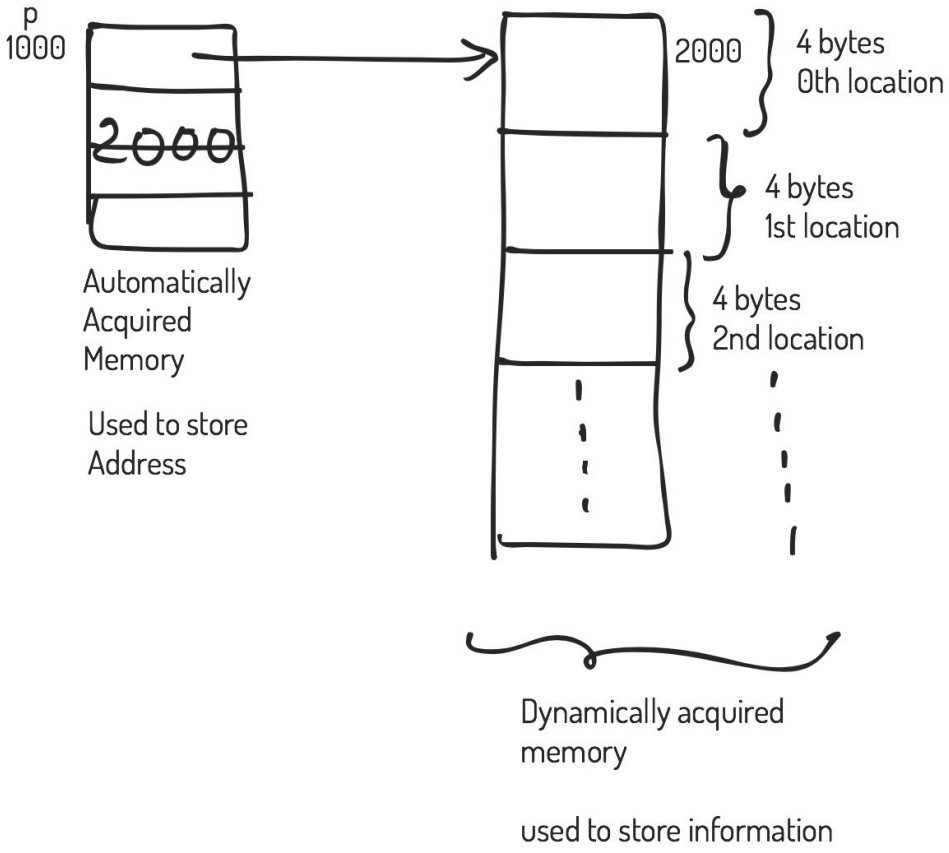
Array - Dynamic creation

Creation of an array in a dynamic manner saves wastage of memory.

```
Ex:    int n, *p;   // n holds a value, which determines the number of information to be stored in array
                    //p holds the base address of the array

       cin>>n;
       p = new int [n];

       for(int i=0;i<n;i++)
            cin>>p[i];

       for(int i=0;i<n;i++)
            cout<<i[p];

       delete [ ]p
```

C++ language - Dynamic memory allocation

p = new int[n];



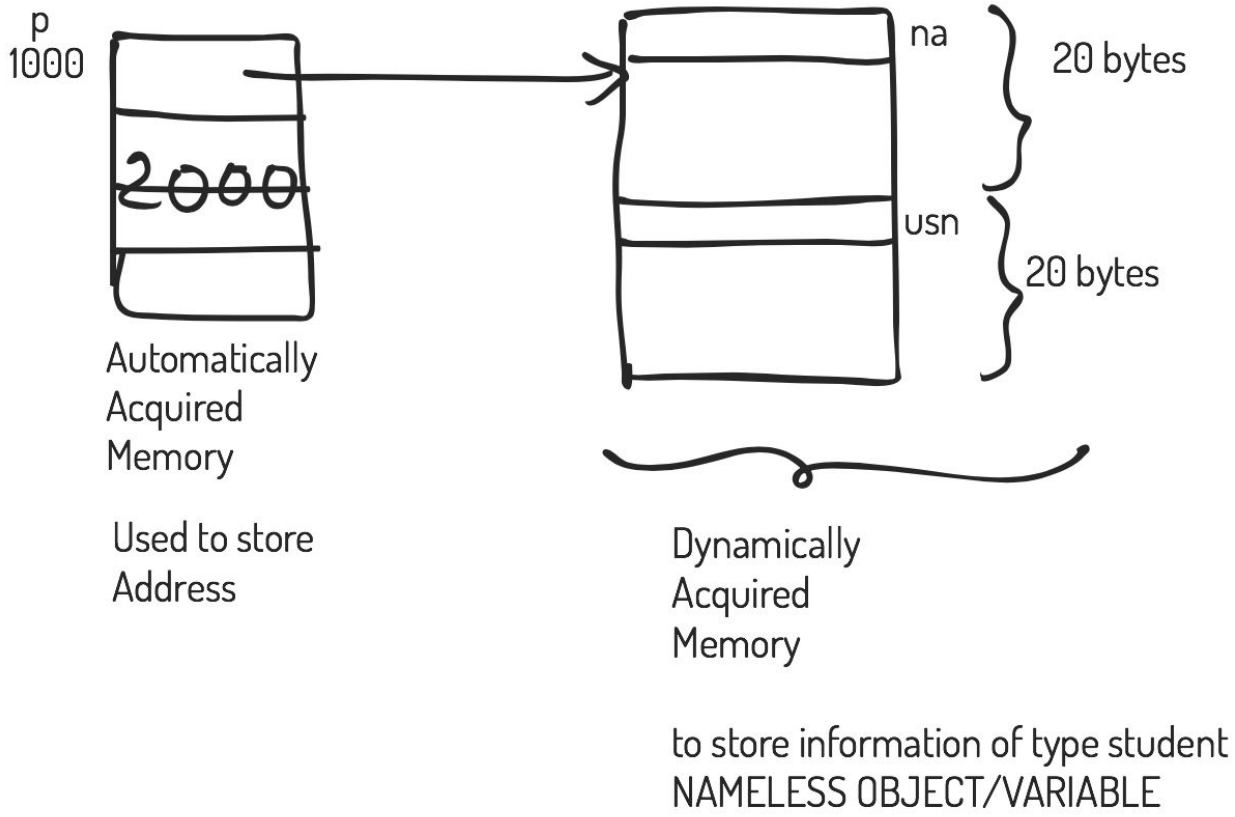
C++ language - Dynamic memory allocation on class types

Ex:

```
class student {
        private:   char na[20], usn[20];
        public:
                        void accept( ) { cin>>na>>usn; }
                        void display( ) { cout<<na<<" "<<usn<<endl; }
};

int  main( )   {
                student  *p = new student;

                (*p).accept( );   p->display( );  delete p;

                int n;   cin>>n;

                p = new student[n];

                for(int i=0;i<n;i++)
                        i[p].accept();

                for(int i=0;i<n;i++)
                        p[i].dispaly();      delete [ ]p;
}
```

C++ language - Dynamic memory allocation on class types

*p = new student;



C++ language - Dynamic memory allocation on class types

p = new student[n];