CONCEPT OF STRUCTS.

```
const dishsize = 2;
 const size = 2;
struct restaurant
{ int choice;
    string dishes[dishsize] = {"ChickenBiryani", "Nihari"};
     string chefs [size]= {"Zubaida", "Batlura"};
    void dish_selection()
    { cout << "select the dish(es) you want to have" << endl;
      for(int i=0; i< size ; i++)
     { cout<< "i+1" << " ." << dishes[i] <<endl;
     }
      cin>>choice;
    }
};
int main()
{
   restaurant obj;
   obj. dish_selection();
   // calling variables.
   for (int i =0 ; i < size; i++)
    {
      chef<<"i+i" << " ." <<objchefs[i] <<endl;
    }
}
```

C++ 11 allows use of public and private access specifiers in structs.
Default for struct : public & default for class : private

CONCEPT OF SETTER FUNCTION:

```
class test
 {private:
   int page;
   float price;
   char name[30];
public:
 void set_page(int p)
{
  cout<<"Entered the no. of
        pages in the
        book".<<endl;
   page=P; }

void set_price(float p)
 {
 cout<<"Entered the price
       of your Book!"<<endl;
   Eit
 price = P;
 }
void set_name (char *n)
{ cout << "Entered the
        name of your book"<<endl;
   strcpy (name, n); }
```

```
int main()
 { test obj;
   obj.setpage(58);
   obj.setprice(1200.28);
   obj.name("The champ!");
}
```

— x —

Concept OF GETTER FUNCT. (store in a variable to display).

```
class test {
   ↓

int get_page(){
  return page; }

float get_price{
   return price; }

string get_name
  { return name; }
 ↓
  };
```

```
+main()
int retrive_page()= obj.get_page();
float retrive_price()= obj.get_price();
string retrive_name()= obj.get_name();
cout << "The book price is" << retrive_price <<endl;
cout<< "The book name is << retrive_pame <<endl;
cout << "The pages of the book"<< retrive_page <<endl;
}
```

CONSTRUCTOR AND ITS TYPES:-

- same name as of class
- no return type
- initialize variables.

Remember Making a parameterized const would require a default const too.

DEFAULT CONSTRUCTOR:-

```
class test{
  public:
  test()
 { cout << "This is default
          constructer "<<endl;
  }
};
```

PARAMETERIZIED CONSTRUCTOR:-

```
class test {
  public
  test (int X, float Y, string a)
  {
  cout << "age:" << X <<endl;
  cout<<"weight:"<< Y <<"\n";
  cout<< "name:" = a <<"\n";
  }
```

CORY CONSTRUCTOR:-

```
#include <iostream>
 using name space std;
class test { private:
             int a;
public
  test(){ cout<<"Default";
          cout<<endl;}
test(test &obj)
{  a = obj.a;
```

```
cout<<a <<endl;
}

test ( int a1)
 { a = a1;
   cout << a <<endl;
 }

int main()
 {
    test obj;
    test obj2(12);

    test z1 (obj 2);
```

IMPL

POINTERS USE IN STRUCT

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

struct student
 {
    char name [80];
    int age;
    float percentage;
 };

  int main()
{
   student obj;
   student *obj1;
    *obj1= &obj;
   obj.age = 1234;
   obj1 -> percentage=13.

   cout<< obj.age<<endl;
   cout<< obj1-> percentage
        <<endl;
```

Pynamically

```
#include <iostream>
using namespace std;
struct student
 {
   int rollno;
   string gender;
 };
int main()
{
student * obj;
  obj = new student;
  obj-> gender= "Male";
cout << obj -> gender;
 delete obj;
}
```

output: male

INTERS USE IN CLASSES:-

The following syntax gives errors if data member in public. (i.e private) not-

dynamically

```
class student
{ public:
    int rollno;
    char gender;
};
int main()
{
  student * obj;
    obj = new student;
  obj -> gender = "M";
  obj -> roll no = 428;
  cout << obj -> roll no;
   cout << endl;
   cout << obj -> gender;

  delete obj;
}
```

non-dynamically

```
class student
{ public
    int age;
    char gender;
};
int main()
{ student obj1
  student *obj2
```

pointer to obj.

```
*obj2 = &obj1;

obj2 -> age = 12;

obj2 -> gender = "F";

cout << obj -> age;
 cout << endl;

cout << obj -> gender;
cout << endl;

}.
```

SCOPE RESOLUTION OPERATOR.

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

class test
 {
  private:
   int a;
   public:
  void s();
 };

void test :: s () {

  cout << "Enter any value to store";
 cin >> a;
 cout << endl;
cout << "you entered" << endl;
```

```
cout << a;
 }
int main()
{
  test t;
  t.s();
  return 0;
}
```

POINTER SIMPLE USE:

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

void swap(int *a, int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;

cout << "The first entry now is"
     << *a << endl;

cout << "The second entry now
     is" << *b << endl;
}
```

```
int main()
 {
  int input1;
  int input2;

cout << "Enter the first
     input" << endl;
  cin >> input1;

cout << "Enter the
     second input"
     << endl;
  cin >> input2;
cout << endl;

cout << "The first entry
     is" << endl;
cout << input1;

cout << "The second
     entry is" << end
 cout << input2;

swap(&input1, &input2
 return 0;
}
```

(to store values of data member using funct use & operator in parametres of funct definintard and prototype.

FRIEND FUNCTION :-

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

class A
{ private:
    int a;
    public:

    friend void get_values (A&, B&);
    friend void add (A&, B&);
};
class B
{ private:
    int a;
  public:
   friend void get_values(A&, B&);
   friend void add (A&, B&);
};

void get_values (A &obj1, B& obj2)
{
   cout << "Enter the value of the first class's object"<<endl;
   cin >> obj1.a;
   cout << "Enter the value of the second classe's
     object" <<endl;
   cin >> obj2.a;
 };

void add (A &obj1, B &obj2)
{
 cout << "The sum of both
     object's certain
     values are " <<endl;
   cout<< obj1.a + obj2.a;
};
int main()
A obj1;
B obj2;
 getvalues(obj1, obj2);
   add (obj1, obj2);
```

Dynamic MA & 2D array!

```
# include <iostream>
using name space std;

int main ()
{

  cout << "Enter the number of rows" <<endl;
   cin >>rows;
  cout << "Enter the number of coloumns" <<endl;
  cin >> cols;

   int **array = new int [rows];
  for ( int i = 0 ; i < rows/cols; i++)
     {  array [i] =  new int [cols];
     }

for (int i =0 ; i<rows ; i++)
 {  for (int j=0 ; j< cols ; i ++)
   {
   cout << "Enter the value for row" <<i+1 <<
           "Enter the value for coloumn" <<j+1 <<endl;

cin>> array[i][j];
   }
} cout <<endl;
}
```

```
// printing the 2D array initialized.
for (int i =0 ; i<raws ; i++)
{
    for (int j=0 ; j<cols ; j++)
    {
        cout << array [i][j] << "  ";
    }
    cout<<endl;
}
// deallocating the memory alloted.
for (int i=0 ; i< rows ; i++)
{
    delete [] array[i];
}
delete array;
return 0;
}
```

imp!

DYNAMIC MA & 1D ARRAY!

```
# Include <iostream>
using name space std;

int main()

{  int b =10; // size of array
   int *a = new int [b];
                // initilizied array.
   for (int i =0 ; i< 10 ; i ++)
   {

      cin>> a[i] ;
      cout<<endl;
   }

delete [ ] a;
   return 0;
   }
```

this pointer!

: data member that store a value equivalent for all the instances created, has a scope of local; lifetime of global.

STATIC DATA MEMBERS!

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

class test
{ private:
static int n;
    public:
     test()
     { n++
     } void display()
{ cout<<n; }
};

int test :: n = 0;
   int main()

{
  test obj, obj2, obj3;
 obj.display();
return0;
}
```

output : 3

STATIC DATA FUNCTIONS!

→ used to access static data members

→ cannot access non-static data members directly, need an obj

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

class test
{
private:
static int num;
  int num2;
public:
static void display()
 {
   cout<< num <<endl;
   num =10;
   cout << "Now:" <<
    num <<endl;
}};

int main()
{ test obj;
  obj.display()
}
```

```
static void display (test& obj)
{ cout<< num;
  cout << obj.num2
}
```

TYPE DEF

A keyword that is used to provide exisiting data types a new name.

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

typedef int myint;

int main()
{
  myint x;
   x =9;
   cout << x;
  return 0;
}
```

—x—

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

typedef struct cat
 {
  string name;
  string food;
  int size;
 }; ep
```

```
int main()
{
  ep obj;
cout << "Enter name" << endl
// cin >> obj.name;
// (creates a space input
   buffer).
  getline (cin, obj.name);
cout " The name entered
        is " <<endl;
 cout << obj.name.
}
```

UNIONS :-

Similar to struct but with better memory management.

```
struct money
{
  int rice;   //2/4 bytes
  char car;  //  1 bytes
  float pound; //  4 bytes
}
```

struct will allocate a total of 9 bytes so that every variable can be called at any time.

```
Union Money
{
  int rice;
  char car;
  float pound;
};
```

Union will allocate memory equal to only the highest 4 bytes that each variable will use at a single time, when active.

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

union money {
  int rice;
  float pounds;
  char car;
};
```

```
int main()
{ money obj;
 //obj. name
 obj. rice = 12; // non active
 obj. pounds = 12.4; // non active.
 obj. car = 'M'; // active variable

cout << obj.rice <<endl; // undefined
                            behaviour.
cout << obj. pounds <<endl; // undefined
                               behaviour.

cout<< obj. car <<endl; // defined behaviour.
}
```

ENUMERATION.

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

enum class mycolors
 {
    Red,
    Green,
    Blue ' };
```

```
intmain()
{
 mycolors obj = mycolors::Re
switch (obj)
 {
case mycolor::Red:
```

```
cout << " R is selected " <<endl;
    break;
case myColors:: Green :

    cout << "G selected" <<endl;
      break;
    case mycolors :: Blue :

     cout << " B selected " <<endl;
       break;
    }

    return 0 ;
    }
```

2D array in OOP Cpp.

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

class Array 2D.
 private :
      int ** data;
       int row;
        int col;
  public:
        Array 2D ( int row, int col) : row(row), col(col)
{
      data = new int * [row];
     for (int i = 0 ; i<row ; i++)
   {     data[i] = new int [col$];
    }.

     ~ Array2D {
for(int i=0 ; i<rows ; i++) { delet [] data[i] ;}
                    delete [] data; }
```