DATE

4/9/23

5/9/23

20EC555
Data Structures and Algorithms

Overview of C++:

Characteristics of OOP:-

- Emphasis on data rather than procedure
- Program is divided into objects
- Data hiding is possible
- Objects can communicate with each other
- New data & functions can be added easily
- It follows bottom up approach

Benefits of OOP:-

- Through inheritance, increase reusability of code
- working module communicate with one another instead of writing the code from scratch.
- Data hiding ensures secure programs.
- Multiple instances of an object.
- Easy to partition the work using objects.
- Message passing makes interfaces simpler.
- Less Software complexity can be easily managed
- Easily upgraded from small to large systems.

Limitations:-

- Data is not fully accessable for all functions
- It is not possible to define random actions instantaneous - ly
- Data members and member functions can not be seperated, therefore feel inconvinient in writing the program
- For any new operation appropriate changes must reflect in class definition.

DATE

Basics:

1. WAP to read two variables and display the values.

→
```
#include <iostream>
void main(){
 int a{},b{};
std::cout << " Enter two numbers: " ;
 std:: cin >> a >> b;
std::cout << " First number: " << a << "Second number: " << b << std::endl;
 }
```

2. WAP to read a variable & display it's address & content:

```
# include <iostream>
void main( ){
 int a = 10;
 int * P{ };
 P= &a;
 std:: cout << " Address : " << P << "Content: " << *P << std:endl;
}
```

3. WAP to add two numbers using functions.

```
# include <iostream>
int sum (int, int);
using namespace std;

void main ( ) {
int a {10};
int b {20};
int c { };
c= sum (a,b);
cout << " Sum: " << c << endl;
}

int sum (int x, int y){
 return x+y;
}
```


DATE

6/9/23

Basic concepts of OOP:-

* Object
* Class
* Polymorphism
* Inheritance
* Data abstraction & encapsulation
* Dynamic binding
* Data hiding
* Message passing

Object:-

Representation:

* It is a run time entity. Ex: They represent student, table of data, Bank account

Object: Student

Data: Name
USN No.
DOB
Marks

Function: Total marks
avg; percentage
display

Class:- Collection of objects of similar type.

* Set of code can be made user defined data type with the he

of class.

- Objects are variables of type class.
- Any no. of objects can be created belonging to that class.
- Class is a user defined data type consists of private, public, data members and member functions.

* It has two parts:
  (i) Class Head (ii) Class body

Ex:
```
Class Student {
 // body
};
```

Syntax:-

```
class classname
{
  private: variable declaration;
           function declaration;
  public: variables;
          functions;
};
```

Ex:

```
class Student
{ private:
  string name;
  int age;
public:
  display();
};

void main(){
  Student S;
  S.display();
}
```

WAP to display the register no., name and marks of a student using OOPs concept

```
#include <iostream>
#include <string>

class Student
{
  private:
    string name{};
    int reg_no.{};
    float marks{};
  public:
    void get_data(){
      cin >> "Enter name: " >> name >> endl;
      cin >> "Enter reg_no: " >> reg_no;
      cin >> "Enter marks: " >> marks;
```


DATE

```
    void display(){
      cout << " Name: " << name << endl;
      cout << " Reg No: " << reg_no << endl;
      cout << " Marks: " << marks << endl;
    }
};

void main(){
  Student S;
  S.getdata();
  S.display();
}
```

11/9/23

WAP to find the area of a triangle using class and objects.

```
#include <iostream>

class Area{
private:
  float breadth;
  float height;

  void get_breadth(){
    cout << "Enter the breadth: "
    cin >> breadth;
  }
  void get_height(){
    cout << "Enter the height: "
    cin >> height;
  }
  void calc_area(){
    cout << " Area of the triangle: "
      << (0.5 * breadth * height) << endl;
  }};

void main(){
  Area triangle{};
  triangle.get_breadth();
  triangle.get_height();
  triangle.calc_area();
}
```

Defining Member functions:

(i) Inside the class

(ii) Outside the class

```
return type class_name:: function name (argument declaration)
{
  Body of the function;
}
```

Ex:

```
class student {
};
void student:: displaydata()
{
}
void student:: getdata()
{
}
```

Data Encapsulation & Abstraction

- Act of representing essential features without including background details of explanation.
- Classes use the concept of abstraction they are known as Abstract data type.

DATE

Inheritance:-

- Process of acquiring the properties from one class/object to the another class/object
- Idea of reusability
- Add additional features to the existing class.

Base class → Derived class

```
class derived_class : visibility baseclass
{
};
```

WAP to calculate the area of the rectangle using base and derived class:

```
#include <iostream
using namespace std;

class rectangle {
Private:
   float width{};
   float height{};

Public:
   void set width (float w)
   {
      width = w;
   }
   void set height (float h)
   {
      height = h;
   }
};
```

```
Class shape : public rectangle
{
public:
   void float getarea(){
      return height * width
   }
};

Void main(){
   float area{};
   shape rect{};
   rect.setwidth(5);
   rect.setheight(10);
   area = rect.getarea();
   cout<< "Area: " << area << endl.
}
```

12/9/23

WAP to create a class A which consist of two member function set_value & display. Write a derived class B which consist of two member function similar to class A. write a main fn to create object for derived class and accessing the member function by passing value

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

class A{
Private:
   int a;
   int b;
Public:
   void set_value (a_val, b_val)
{
   a = a_val;
   b = b_val;
}
void display (){
   cout<< "The values are a:"
   << a << "b: " << b << endl;
}
};

class B : public A {
   int z;
   B (int k){
      z = k;
   }
   void dis(){
   cout<< "z = " << z << endl;
   }
};

Void main(){
   B obj{5};
   obj. set_value (10, 12);
   obj. display ();

   obj. dis();
}
```

Output:

```
The values are a: 10 b: 12
z = 5
```

WAP to exchange values of two variables by call-by-value & call by reference

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

void call_by_value (int, int);
void call_by_ref (&int&, int&);

void main (){
   int a, b;
   cout<< "Enter the values a& b: "
   cin >> a >> b;
   call_by_value (a, b);
   cout << "a: " <<a << "b: " << b<<endl;
   call_by_ref (a, b);
   cout << "a: " <<a << "b: " << b<< endl;
}
void call_by_value (int x, int y){
   int temp;
   temp = x;
   x = y;
   y = temp; cout << "a: "<<a<<"b: "<<b<<endl
}

void call_by_ref (int &x, int &y){
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Output:

```
Enter the values of a & b: 10 12
a: 12  b: 10
a: 10  b: 12
a: 12  b: 10
```

13/9/23

DATE

WAP to illustrate working of new & delete operator

```
#include <iostream>
using namespace std;
void main(){
int *a = new int;
int *b = new int;
int *sum = new int;
cout<<"Enter a & b: ";
cin>>*a>>*b;
*sum = *a + *b;
cout<<"Sum: "<<*sum;
delete *a;
delete *b;
}
```

19/9/23

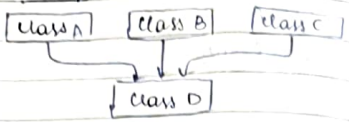
Types of inheritance:

1. Single / simple inheritance
2. Multiple
3. Multilevel
4. Hierarchical
5. Hybrid

1. Single:

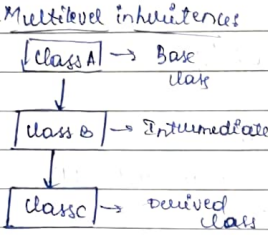
```
class A
{ public: int x,y;
};
class B: public A {
{
public: int a,b;
};
```

2. Multiple inheritance:




```
class A{
public: int x,y;
};
class B {
public: int m,n;
};
class C {
public : int p, q;
};
class D: public class A, class B, class C {
public: int r,s;
};
```

3. Multilevel inheritance:

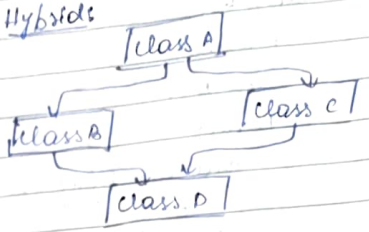



```
class A {
public: int x, y;
};
class B: public class A
{
public: int a,b;
};
class C: public class B
{
public: int z, s;
};
```

4. Hierarchical:


```
class A {
public: int x, y;
};
class B: public class A
{
public: int a,b;
};
class C: public class A
{
public: int i,d;
};
class D: public class A
{
public: int m,n;
};
```

5. Hybrid:




```
class A
{
public: int a,b;
};
class B: public class A
{
public: int c,d;
};
class C: public class A
{
public: int e,f;
};
class D: public class C, class D
{
public: int x,y;
};
```

WAP to illustrate multiple inheritance

```
class A{
private: int a;
public: void get_a(int x){
a = x;
}
};
```

DATE

```
class B {
Private:
  int b{};
Public:
  void get_b(int y){
    b = y;
  }
};

class C : Public A, Public B
{
Private:
  int sum;
Public:
  void sum(){
    sum = a+b;
    cout<< "Sum= "<< sum;
  }
};

void main(){
  C obj;
  obj.get_a();
  obj.get_b();
  obj.sum();
}
```

Illustrate Hierarchical inheritance

base class consists of one member fn to read two variables

```
#include <iostream>
using namespace std;
class A
{
Public:
  int x,y;
  void get_numbers(){
    cout << "Enter two numbers: ";
    cin >> x >> y;
  }
};

class B : Public A {
Public: int p;
  void Product(){
    p = x*y;
    cout<< "Product is: "<<p<<endl;
  }
};

class C : Public A {
Public: int sum;
  void sum(){
    sum = x+y;
    cout<< "Sum: "<< sum<<endl;
  }
};
void main(){
  B p;
  C s;
```

Output:

Enter a number: 2
Enter another number 3
Sum = 5

```
  p.get_numbers();
  p.Product();
  s.get_numbers();
  s.sum();
}
```

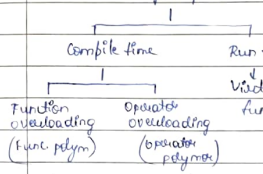
Output:

Enter two numbers: 5 4
Product is: 20
Enter two numbers: 5 4
Sum: 9

Polymorphism:-

- Ability to take more than one form.

Function overloading:-



- Process of making an operator behave differently in different instances is operator overloading.

WAP to illustrate Polymorphism.

```
#include <stdiostream>
using namespace std;
void main(){
  int a, b, sum;
```

```
#include <>
#include " "
```

DATE

```
  float x, y, total;
  double m, n, gross;
  cout << "Enter the value for a,b,x,y,m&n"<<endl;
  cin >> a >> b >> x >> y >> m >> n;
  sum = add(a,b);
  total = add(x,y);
  gross = add(m,n);
  cout<< "Sum = "<<sum<<endl;
  cout<< "Total = "<< total<<endl;
  cout<< "Gross = "<< gross<<endl;
}
int void add(int a, int b){
    return (a+b);}
float add(float a, float b){return (a+b);}
double add(double a, double b){return (a+b);}
```

- Fn is taking many forms depending on data type.
- Fn behaves differently based on parameters given.

Using Classes:-

```
#include <iostream>
using namespace std;
class Display {
Private:
  int a,b;
  float x,y;
  double m,n;
Public:
  void display(int a, int b){
    cout << "Values are: "<<a<< " "<<b
  }
  void display(float x, float y){
    cout<<"Values are: "<<x<<" "<<y<<
```

(int a, int b) and (float x, float y) not required

```
void disp (double m, double n){
  cout<<"Value of m: "<<m<<" n: "<<n<<endl;
}
};

void main (){
  Display d;
  d.disp (4.5);
  d.disp (10.15, 11.54);
  d.disp (3.141576, 5.536274);
}
```

Operator overloading:-

- Defining the additional tasks to operators without changing actual meaning
- To perform different operations on the same operands
- To provide the special meaning to the user defined data types

Syntax:

```
return_type classname operator symbol (classname args)
  {
    ---
  }
```

```
#include <iostream>
class complex {
  private:
    int real, img;
  public:
    Complex (int real=0, int img=0): real(real), img(img)
  {};
```

classmate PAGE

```
void print (){
  cout << real << "+j" << img << endl;
};

int Complex operator + (Complex & obj)
  {
    Complex res;
    res.real = real + obj.real;
    res.img = img + obj.img;
    return res;
  }
};

void main (){
  Complex N1(10.5), N2(2.4);
  Complex N3 = N1+N2
  N3.print();
}
```

Output:

12 + i9

// Concatenate two strings.

```
#include <iostream>
#include <string>
```

classmate PAGE

two strings.

```
#include <iostream>
#include <string>
```

Virtual functions:-

* ~~Virtual fn~~ WAP to illustrate virtual functions.
* Run time polymorphism.
* Uses keyword virtual
* They must be a member of some class
* Virtual fn can be a friend of another class
* They shd be declared in public section of class.
* They can't be static member function
* They can be accessed by using base pointer & object
* Prototype of base class version of a virtual function & all the derived class prototype shd be identical.

DATE

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

class baseClass {
  Public:
    virtual void virtfunc() {
     cout << " This is baseclass function " << endl;
    }
};

class derivedclass: public baseClass {
  Public:
     virtual void virtfunc() {
      cout << " This is derived class function " << endl;
     }
};

void main() {
 baseClass *ptr, b;
 derivedclass d;
 ptr = &b;
 ptr -> virtfunc();
 ptr = &d;
 ptr -> virtfunc();
}
```

Output:

This is baseclass function.
This is derived class function

DATE

Inline function: It is a function that is expanded inline when it is invoked.

Syntax:

```
inline function header
{
  function body;
}
```

Ex:
```
inline int cube (int a) {
return (a*a*a);
}
```

WAP to find largest of 3 nos

```
(a>b) ? ((a>c) ? a : c) : ((b>c) ? b : c)
```

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

void main () {
int a, b, c;
cout << " Enter three nos: ";
cin >> a >> b >> c;
large (a, b, c);
}

inline int large (int x, int y, int z) {
  if (x>y)
    big = x;
  else big = y;
  if (z>big)
    big = z;
  cout << "The largest = " << big << endl;
}
```

NOTE: Inline functions cannot return back anything

DATE

5/10/23

Friend function :-

* Declaring the non-member functions as friend functions to the class to access the data members of the class.

Ex:
```
class ABC{
Public:
 friend void xyz();
};

void xyz(){
 // Body of the function
}
```

* The friend fn declaration shd be preceded by the keyword friend
* A friend function is a normall C++ func.
* A friend func can be declared in private / public section of the class.
* Usually friend fns has objects as its arguments
* A friend fn cannot access class members directly, an object name followed by dot operator is used.
* Member fn of ~~One~~ one class can be a friend of another class.
* One class can be a friend of another class.

WAP to illustrate friend fn.

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

class sample {
private: int a,b;
public: void get_data(){
            a=25; b=40;}
friend float mean(sample s);
};
```

classmate

PAGE

```
float mean (sample S){
    return (S.a + S.b)/2.0;

void main (){
  sample x;
   x. get_data ();
  (x.mean(x);)x
  cout<< " Mean is : " << mean (x) << endl;
}
```

Output:

Mean is : 32.5

Member fn of one class friend of another class

```
class x {
  int func1 ();
};

class y {
  friend int x:: func1();
}

class z {
  friend class x;
};
```

Constructors and Destructors :-

* Automatic initialization of objects.
* It is a special member fn whose name is same as class name
*

```
Ex: class Sample{
    int m, n;
    public : Sample (){
            m=0 ; n=0;
        }
    };
```

DATE

Types:-

1. Default constructor
2. parameterized - " -
3. copy constructor
4. Dynamic constructor
5. Overloaded constructor

→ Default constructor

* that accpts no parameter

sample a; // invokes the default constructor.

```
class box {
  int len, breadth, height;
  public: void box() {
            length = breadth = height = 0; }

  void print(){
   cout << "length: " << len;
   cout << "breadth: " << breadth;
   cout << " height: " << height;
  }
};

void main() {
  box a;
  a.print();
}
```

Output:

length = 0;

breadth = 0;

height = 0;