OOP x AI

Created	@July 26, 2023 7:52 PM
Last edited time	@July 31, 2023 9:52 PM
Created by	B Borhan
Tags	OOP Year 2 Term 1

Some Basics

Namespace

- allows to organize code elements, avoiding naming conflicts/collision when the code base include multiple libraries

:: → Scope Operator/Scope Resolution Operator

Structure	Class
If access specifier is not declared explicitly, then by default access specifier will be public.	If access specifier is not declared explicitly, then by default access specifier will be private.
Syntax of Structure struct structure_name{// body of the structure.}	Syntax of Class class class_name{// body of the class.}
The instance of the structure is known as "Structure variable".	The instance of the class is known as "Object of the class".

OOP : Object Oriented Programming

- a programming paradigm
- it allows decomposition of a problem into number of entities called “objects” and then builds data and function around this objects
- Feature:
    - Emphasize on data rather than procedure
    - Programs are divided into “object”
    - Data is hidden and cannot accessed by external functions
    - Object my communicate with each other

    - New data and function can be easily added

Class

- a class is a blueprint or template that define the structure and behavior of objects
- it serves as a user define data type
- **Members of a class**
    - Variable → data member
    - Function → member function/methods
- **Access Specifiers**
    - **Public:** Can be accessed from publicly/outside of class too
    - **Private:** Can be accessed from inside the class
    - **Protected:** Can be access from inside the class and the inherited class

```
class myClass{
  int z; // private

  public :
    int x, y; // data member

    // member function
    int setter(int a, int b){
      x=a; y=b;
    }

    int show(){
        cout << x+y << "\n";
    }

    int functionCanBeWrittenFromOutside();
};

int myClass::functionCanBeWrittenFromOutside(){
  return x;
}
```

Objects

- an object is an instance of class that combines data and behavior

- can be created using new “operators”
- an object occupies memory space
- Private and Protected members can’t be accessed from object (a compile time error) , but public members

```
myClass ob1;
ob1.setter(1,2);
ob1.show();
```

- **Passing object**
    - Constructor function is not called, because the object is not initializing but copying
    - the function terminates and the copy destroyed, the destructor function is called

```
class myClass{
....
public:
int x;
...
};
int sq(myClass ob){
  return ob.x * ob.x ;
}
int main(){
...
  myClass ob1, ob2;
  ...
  cout << sq(ob1) << "\n";
..
}
```

- Returning Object from Functions

```
class myClass{
...
};

myclass func(){
  myClass ob1;
  ...
  ...
  return ob1;
}

int main(){
```

```
..
myClass ob;
ob = func();
..
}
```

- **Array of Objects**

```

class yourClass{
int x;
...
public :
myClass(int a){
  x = a;
};
...
};

class myClass{
int x, y, z;
...
public :
myClass(int a, int b, int c){
  x = a; y=b; z=c;
};
...
};

int main(){
..
// one dimensional array of object with one argument
yourClass ob1[3] = {1, 2, 3};
// or
yourClass ob1[3] = {yourClass(1), yourClass(2), yourClass(3)};
// accessing
ob1[0].show();

// one dimensional array of object with more than one argument
myClass ob1[3] = {yourClass(1,2,3), yourClass(2,3,1), yourClass(3,1,2)};


// multi-dimentsional array of object
yourClass b1[2][2] = {
  yourClass(1), yourClass(2),
  yourClass(3), yourClass(4),
};
// accessing
b1[0][0].show();
...
}
```

- Using Pointers to Objects

- Same as other data type. If a pointer increments, it points the next object. If decrements → point the previous object.

  ```
  class myClass{
  ...
  ...
  };

  int main(){
    ...
    myClass ob[4] = {1,2,3,4};
    myClass *ptr;
    ptr = ob;
    for(int i=0; i<=3; i++){
      cout << p->show() << "\n";
      p++;
    }
    ..
  }
  ```

- *this* pointer
  - ***This*** is a keyword that is used to represent an object that invokes (call, `Āhbāna`) the member function
  - a pointer that is **automatically passed** to any member function when it is called, it is a **pointer to the object that generates** the call
  - **Syntax:**
    - Syntax for referring instance variable

      ```
      this->data_member = value;
      ```

    - Syntax for referring to current object of class

      ```
      this*
      ```

    - **Example:**

    ```
    class myClass{
    int x, y, z;

    public:
    void set(int a, int b, int c){
      this->x = a;
      this->y = b;
    ```

```
  this->z = c;
}
...
};
```

- **Applications**
  - **Distinguish data members from local variables**

    Example: Distinguishing between the class members and the parameters with same name.

```
class myClass{
  int x, y, z;
  public:
  void set(int x, int y, int z){
    this->x = x;
    this->y = y;
    this->z = z;
  ..
  }
...
};
```

  - **Method Chaining**

    Method chaining is a very useful feature of this pointer. It helps in chaining the method togethers for erase and cleaner code.

    Example: `obj→set(10)→replace→(9)→print();`

```
#include<bits/stdc++.h>
using namespace std;

class myClass{
  int x;

  public:
  myClass & set(int x){
    this->x = x;
    return *this;
  }

  myClass &replace(int x){
    this->x = x;
    return *this;
  }

  void print(){
    cout << this->x << " ";
    }

  };

  int main(){
    myClass obj;
    obj.set(10).replace(20).print();
  }
```

- **Allocating and deallocating memory**
  - Allocating memory

```
// Syntax
type *var_name = new type;

// Variable pointer
int *ptr = new int;
char *ptr = new char;

// Giviing initialed value
int *ptr = new int(8);

// Checking allocation error
if(!ptr){
  cout << "Allocation error";
}

// One Dimension Array
int *ptr = new int[100];

// Two-dimensional array
int row=10, col=10;
int **table = new int*[rows];
for(int i=0; i<row; i++){
  table[i] = new int[col];
}
```

  - Deallocating memory

```
//Syntax
delete ptr;
delete [] ptr; // array
ptr = NULL;

// Variable
delete ptr;
ptr = NULL;

// One-dimensional array
delete [] ptr;
ptr = NULL;

// Two Dimensional Array
```

```
for(int i=0; i<row; i++){
  delete [] table[row];
}
delete[] table;
table = NULL;
```

  - If we create an **array of objects,** then destructor will be called when we deallocate or delete or free the array of object.
  - **Choosing new/delete over malloc/free**
    - Type safety: automatically call constructor and destructor
    - Easier memory management : no need to calculate memory
    - No need to explicit type casting

Example:

- Create a class that contains a person's **name** and **telephone number**. **Using** `new`, dynamically allocate an object of this class and put your name and phone number into these fields within this object.

```
#include<bits/stdc++.h>
using namespace std;

class Person{
  string name;
  string telephone;
  public:
  Person(string name, string telephone){
    this->name = name;
    this->telephone = telephone;
  }
  void show(){
    cout << "Name: " << this->name << "\n" << "Telehpone: " << this->telephone;
  }
};

int main(){
  string name, telephone; cin >> name >> telephone;
  Person *p = new Person(name, telephone);
  p->show();
}
```

- **References**
  - A reference can be defined as an alternative name of a variable

- **Sign** : & (ampersand)
- **Syntax:**

  ```
  int data = 10;
  int &ref = data;
  ```

- **Reference can be used in 3 ways**
  - **Independent Reference**

    An independent reference is a reference variable that in all effects is simply another name for another variable

    - We cannot reference another reference
    - We cannot obtain the address of a reference
    - We cannot create arrays of reference

    ```
    int main(){
      int data = 10;
      int &ref = data;
      cout << ref << "\n"; // Output:10
    }
    ```

  - **Passing reference to the function as parameter/Call-by-reference**

    While passing the reference to the function, the changes made inside the function will also be reflected/changes outside.

    ```
    void increment(int &n){
      n++;
    }
    int main(){
      int x=5;
      increment(x);
      cout << x << "\n"; // Output :6
    }
    ```

  - **Returning Reference**
    - While returning the reference an implicit pointer is also returned

        - the returning variable must be global/static
        - no local and constant variable is returned
        - function calling is done on the left-hand side of the assignment operator for assigning value

        ```
        #include <bits/stdc++.h>
        using namespace std;

        int n;

        int &func(){
          return n;
        }

        int main() {
          // assigning value
          func() = 10;

          //getting the value
          int *p;
          p = &func();

          //getting the value by reference
          int &ref = func();

          cout << ref << " " <<  *p <<  "\n";
        }
        ```

    - **Advantages of Reference**
        - It helps modifying or changing the values inside function without passing the actual arguments
        - It helps with operator overloading
        - It helps with writing less error-prone code
        - Avoid unnecessary copy of data
    - **Example: Swap two numbers using reference.**

        ```
        void swapp(int &a, int &b){
          t=a;
          a=b;
          b=t;
        }
        int main(){
          int x=8, y=80;
          swapp(x, y);
          cout << x << "  " << y << "\n";
        }
        ```

- **Passing reference to Objects**
    - passing object by call-by-value → a copy of the object is made → constructor isn't called but **destructor called** when function returns
    - passing object call-by-reference → no copy is made → constructor isn't called and destructor isn't called when function returns

```
class myClass{
......
};

int func(myClass &ob){
  ...
  ...
}

int main(){
  ....
  myClass obj;
  int x = func(obj);
  ....
}
```

Constructor

- a special member function of a class that is **automatically called when an object of that class is created**
- **Rules**
    - No return Type
    - Same name as the class
- **Types**
    - **Default/Non-Parameterized** : constructor function with no arguments/parameter
    - **Parameterized Construction:** constructor function with arguments
    - **Copy Constructor:** a constructor that creates a new object by initializing it with an existing object of the same class
        - initializing with an existing object

- passing object as function argument
- using the class as a return-type of any function

```
//Default or Non-parameterized Constructor
class amarClass{
  int x;
  public :
  amarClass(){
    cout << "Constructor" << "\n";
  }
};
int main(){
  amarClass a;
}


// Parameterized Constructor
class apnarClass{
  int x;
  public :
  //01
  apnarClass(int a, int b){
    x = a;
    cout << "Constructor " << a << " " << b << "\n";
  }
  //02 Using default arguments
  apnarClass(int a=0){
    x = a;
    cout << a << " ";
  }
};
int main(){
  apnarClass b(8, 80);
  apnarClass c;  // 02
}

//Copy Constructor
class array{
  int *p;
  int size;
  public:
...
    array(int sz){
        p = new int[sz];
        if(!p){
            exit(1);
        }
        size = sz;
    }
    array(const array &a){
      p = new int[a.size];
      if(!p) exit(1);
      size = a.size;
      for(int i=0; i<size) i++){
        p[i]=a.p[i];
      }
    }
    void put(int i, int j){
      if(i >= 0 && i < size){
          p[i]=j;
      }
    }
....
};
int main(){
...
  array obj(4);
  for(int i=0;' i<4; i++) put(i, i+8);
  array obj2(obj); // copy constructor
....
}
```

- **For Global Objects** :
  - declaring object globally (before main function)
  - object’s constructor is called once, when program first begins execution

  ```
  class tomarClass{
  ....
  };
  tomarClass tumi; // declaring globally
  int main(){
  }
  ```

- **For Local Objects:**
  - the constructor is called each time, the declaration statement execute

  ```
  class tomarClass{
    .....
  };

  int main(){
    tomarClass tumi; // declaring locally
  }
  ```

- **Overloading Constructor Function**
  - **Overloading Function:** multiple functions with the same name to be defined in the same scope but with different parameter lists
  - **Reasons**
    - to gain flexibility
    - to support array

        - to create copy constructor

```
class myClass{
  int x;
  public:
...
  myClass(){ x = 0; }
  myClass(int a) { x = a; }
  myClass(int a, int b) { x=a*b;}
  .....
};

int main(){
...
myClass a, b(1), c(1, 2);
...
}
```

- **Assignment operation**

```
class myClass{
.....
};

int main(){
  myClass a, b;
  ...
  b = a;
}
```

Destructor

- a special member function of class that is invoked automatically when an **object of that class is about to be destroyed**
- **Rules**
    - No return type
    - Same name as class
    - a `tilde (~)` sign before name of the class

```
class myClass{
  public :
  ~myClass(){
    cout << "Destructor" << "\n";
  };
};
```

- Calling
    - Implicitly Called by the compiler when an object goes out of scope
    - Explicitly deleted using *delete* operator
- Local/Global
    - Local objects are destroyed → they go out of scope
    - Global objects are destroyed → the program ends

Home Work:

- Create a class called **stopwatch** that emulates a stopwatch that keeps track of elapsed time. Use a constructor to initially set the elapsed time to 0. Provide two member functions called **start()** and **stop()** that turn on and off the timer, respectively. Include a member function called **show()** that displays the elapsed time. Also, have the destructor function automatically display elapsed time when a stopwatch object is destroyed. (To simplify, report the time in seconds.)

```
...
#include<ctime>
..
class stopwatch{
  clock_t start, end;
  clock_t object_start = 0;
  double elapsed_time;

  public:
  stopwatch(){
    elapsed_time = 0;
    object_start = clock();
  }
  ~stopwatch(){
      end = clock();
      elapsed_time = (double) (object_start - end)/CLOCK_PER_SECOND;
      cout << elapsed_time << "\n";
  }
  void start(){
    start = clock();
  }

  void stop(){
    stop=clock();
  }
  void show(){
    elapsed_time = (double)(end-start)/CLOCK_PER_SECOND;
    cout << elapsed_time << "\n";
  }
};
```

- Create a class called box whose constructor function is passed three double values, each of which represents the length of one side of a box. Have the box class compute the volume of the box and store the result in a double variable. Include a member function called vol() that displays the volume of each box object.

```
class box(){
  double a, b, c, volume;
  public:
  box(double x, double y, double z){
    a=x; b=y; c=z;
    volume = a*b*c;
  }
  void vol(){
    cout << volume << "\n";
  }
}
```

Inheritance

একটি প্রোগ্রামিং প্রক্রিয়া, যেখানে একটি ক্লাস আরেকটি ক্লাসের বৈশিষ্ট্যগুলি (members function, method) ব্যবহার করতে পারে।

- is the mechanism by which one class can inherit the properties of another class
- **Base Class:** The class from which the child class inherits its properties is called the parent class or base class.
- **Derived Class:** The class that inherits the characteristics of another class is known as the child class or derived class

Access Specifier of Base Class	Public Inheritance	Private Inheritance	Protected Inheritance

Access Specifier of Base Class	**Public Inheritance**	**Private Inheritance**	**Protected Inheritance**
Public	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected

```
class base{
  int x, y, z;
  ..
  public :
  ..
  int sum(){
      z = x+y;
  }
  ...
}

// Public Inheritance
class derived : public base{
....
}

// Private Inheritance
class derived2 : private base{
....
}

// Protected Inheritance
class derived3 : protected base{
....
}
```

- **Constructor in inheritance**
    - The constructor of base class (default or non-parameterized) and derived class called in sequence (first base class and then derived class)
    - To call parameterized-constructors of base class, we have to maintain it explicitly.

    ```
    class base{
    ...
    public:
    base(){
      cout << "Default Constructor from base class\n";
    }
    base(int x){
      cout << x << "\n";
    }
    };
    ```

```
class sub{
...
public:
sub(){
  cout << "Default constructor from sub class\n";
}
sub(int a, int b) : base(b)// calling base(int x) explicitly
{
  .....
}
};
```

  - Destructor are called in a reverse order
    - First, derived class destructor
    - Then, Base class destructor
- **Types of Inheritance**
  - **Single Inheritance**
    - most primitive among all types
    - a single class inherits the properties of base class

```
class base {
...
};

class sub : public base{
.....
};
```

  - **Multiple Inheritance**
    - a class can inherits properties of multiple base classes

```
class base1{
....
};

class base2{
....
};
// 01
class sub : public base1, public base2 {
.....
.....
  //calling parameterized constructor
  sub(int x, int y, int z) : base1(x, y, z), base2(x, y){
  ....
  }
...
};

//02
class sub2 : public base1, private base2 {
.....
.....
  //calling parameterized constructor
  sub2(int x, int y, int z) : base1(x, y, z), base2(x, y){
  ....
  }
...
};
```

- **Constructor Call :** sub → base1 → base2 (check the inheritance order in code)
- **Destructor Call:** base2 → base1 → sub (check the inheritance order in code)

- **Multilevel Inheritance**
  - a derived class created from another derived class

```
class A{
....
};
class B : public A{
....
};
class C : public B{
...
};
```

- **Multipath Inheritance**
  - a class derived from multiple derived class with same base class

```
class A{
...
};
class B : public A{
....
};
class C : public A {
...
};
class D : public C, public B, public A{
```

```
....
};
```

- **Hierarchical Inheritance**
  - **If more than one class is inherited from the base class**, it's known as hierarchical inheritance.

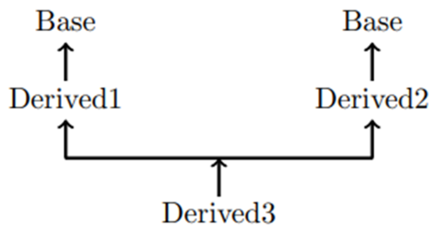
    ```
    class A{
    ...
    };
    class B:public A{
    ...
    };
    class C:public A{
    ...
    };

    class D:public B{
    ...
    }
    class E:public B{
    ....
    }
    class F:public C{
    ...
    };
    class G:public C{
    ....
    }
    ```

- **Hybrid Inheritance**
  - combination of two or more types inheritance

  ```
  // combination of multilevel and hierarchy inheritance
  class A{
  ...
  }
  class B {
  ...
  };
  class C : public A, public B{ // example of multi-level
  ....
  };
  class D :public  A {
  ...
  }
  class E : public D {
  ...
  };
  class F : public D{
  ```

```
....
};
```

- **Virtual Base Class**



Derived3 class called “Base” class for two times to reduce this ambiguity we will use virtual class. There are two path two reach Base. If we use **virtual base class** , it will choose one path automatically. Otherwise, the compiler will visit the both path.

- it generates two copies
- it will give a compiler error (assigning a variable, ..)

```
class base[
...
]
class deriverd1 : virtual public base{
...
}
class derived2 : virtual public base {
...
}
class derived3 : public deriver1, public derived2{
....
}
```

- **Advantages of Inheritance**
  - Reduce code redundancy
  - Code reusability
  - Reduce source code
  - improves code readability
  - code is easy to manage
  - support code extensibility using overriding function
- **Disadvantages of Inheritance**

    - if you change in parent, that will affect in child classes
    - in hierarchy inheritance, many data members and functions are unused → memory not utilized → affect in performance

In-Line Function

- suggest the compiler to replace the function with the actual function definition at the calling site
- the compiler to substitute the code within the function definition for every instance of a function call.
- **Advantages:**
    - **Performance Improvement:** Faster because of eliminating the overhead of function call
    - **Code Size Reduction:** reducing code size by avoiding repeated function call instructions
    - **Avoid Function Call Stack:** do not require pushing and popping function call stack
- **Disadvantages**
    - **Increased Code Size:** If they are too large and called too often, program grown larger
    - **Limited Optimization Opportunities:** In-lining a function call limit the compiler’s ability to optimize the code
    - **Compiler Decision:** The compiler ultimately decides whether to inline a function or not and the “inline” keyword is merely a suggestion

```
inline int cube(int x){
  return x*x*x;
}
int main(){
...
int a=5;
int ans = cube(a);
....
}
```

- A function convert to inline automatically (without *inline* keyword)
  - the function is defined in a class
  - the function's definition is short enough

- Create a class called **dice** that contains one private integer variable. Create a function called **roll()** that uses the standard random number generator, **rand(),** to generate a number between **1 and 6**. Then have **roll()** display that value.

```
#include <cstdlib>
#include <iostream>
#include <time.h>
...

class dice(){
  int val;
  public:
  void roll(){
    srand(time(0));
    val = rand()%6 + 1;
    cout << roll << "\n";
  }
}

...
```

Friend Functions

- a friend function is a function that is granted access to the private and protected members of a class, even though **it is not a member of that class**
- keyword : $friend$
- it doesn't have **this** pointer
- **Reasons :**
  - operator overloading
  - creation of certain type of I/O functions
  - accessing private and protected member of a class without being member of that class

```
class myClass{
...
int x; // x is a private data member
...
public :
...

friend int isEven(myclass ob);
..
};

int isEven(myclass ob){
  return (ob.x%2==0);
}

int main(){
...
myClass obj;
...
cout << isEven(obj) << "\n";
...
}
```

- **Moreover**
  - a friend function can work with two or more different classes too

    ```
    class truct;

    class car{
    int speed:
    ...
    public:
    friend int sp(car c, truct t);
    ...
    };

    class truct{
    int speed;
    public:
    friend int sp(car c, truct t);
    };

    int sp(car c, truct t){
      ..
      return c.speed >= t.speed;
    }
    ....
    ```

  - a friend function can be member of other class

    ```
    class car;

    class truct{
    ...
    public:
    ```

```
int friend_func(car c, truct t);
...
};

class car{
...
public:
friend int car::friend_func(car c, truct t);
};

int car::friend_func(car c, truct t){
.....
}
```

Operator Overloading

- allows you to redefine the behavior of standard operation (+,-,*,/, ++ ..) for user-defined data types
- to overload operator, create an operator function
- most often an operator function is a member function or friend of the class
- **Syntax:**

```
class .. {
  return_type operator symbol (arguments){
    ...
  }
}

symbol : +,-,++, ...
```

- **Restrictions**
  - Precedence of operator cannot be changed
    - Operator precedence ***determines the grouping of terms in an expression and decides how an expression is evaluated***
  - The number of operands that an operator takes cannot be altered
    - The statement "The number of operands that an operator takes cannot be altered" means that you cannot change the number of operands (arguments) that an operator works with when overloading it in C++. Each

operator has a fixed number of operands it operates on, and this cannot be modified during operator overloading.

For example:

- The binary arithmetic operators (+, -, *, /) take two operands.
- The unary arithmetic operators (+, -) take one operand.
- The comparison operators (==, !=, <, >, <=, >=) take two operands.
- The logical operators (&&, ||) take two operands and use short-circuit evaluation.
- The assignment operator (=) takes two operands, one on the left and one on the right.

- Operator function cannot have default arguments

- **Binary Operator Overloading**
    - **Binary Operator:** Binary operators are operates in programming language that operates on two operands or values
        - Example: + ⇒ a+b ⇒ it needs two operands *a* and *b*
    - When member function overloads a binary operator, the function will **have only one parameter → the right operand of operator**
    - **the left operand of operator → generates the call → passed implicitly by *this***

```
class myClass{
  int x;
  public:
  ...
  // right side can be an object
  myClass operator+(myClass ob2){
    myClass temp;
    temp.x = x + ob2.x; // x = this->x
    return temp;
  }

  // right side can be an interger
  myClass operator+(int i){
    myClass temp;
    temp.x = x+i;
    return temp;
  }

// return integer, boolean,..
bool operator==(myClass ob2){
  return x==ob2.x;
}
  ...
}
int main(){

myClass a, b, c;
...
c = a+b;
...
c = a+1;
/* c= 1+a => it will give compile-time error,
 friend function is needed to solve this */
}
```

- **Unary Operator Overloading**
  - **Unary Operator:** operates on one operand only
    - Example: ++, - -
  - the operator function has no parameters
  - the operands ⇒ generates the call

```
//obj++;
class myClass{
  int x;
  public:
  ...
  myClass operator++(){
    x++;
    return *this;
  }
  ...
}

//++obj; notused value always be 0
...
myClass operator++(int notused){
  x++;
  return *this;
}
...
```

- **Friend Operator Function**
  - **Friend Function:** a friend function is a function that is granted access to the private and protected members of a class, even though **it is not a member of that class.**

- a friend function dose not have a **this** pointer
- a friend operator function is passed **two(for binary) / one(for unary)** operands explicitly (reason : there’s no **this** pointer)

```
class myClass{
  int x;
  public:
  ..
  friend myClass operator+(myClass o1, myClass 02); // c=a+b
  friend myClass operator+(int i, myClass o1); // c = 1 + a
  friend myClass operator+(myClass o1, int i); // c=a+1
  friend myClass operator--(myClass o1); // a--
  ..
};
// c=a+b
myClass operator+(myClass o1, myClass o2){
  myClass temp;
  temp.x = o1.x + o1.x;
  return temp;
}
//c=1+a
myClass operator+(int i, myClass o1){
  myClass temp;
  temp.x = i + o1.x;
  return temp;
}
//c=a+1
myClass operator+(myClass o1, int i){
  myClass temp;
  temp.x = i + o1.x;
  return temp;
}
//a--
myClass operator--(myClass o1){
  o1.x--;
  return o1;
}
```

Exception Handling

- a mechanism that allows to manage and respond to unexpected or exceptional situation that may occur during the executing time
- handling run-time error
- **Types**
  - **Checked :** Also called *compile-time exceptions*, the compiler checks these exceptions during the compilation process to confirm if the exception is

being handled by the programmer. If not, then a compilation error displays on the system.

- **Unchecked:** Also called *runtime exceptions*, these exceptions occur during program execution. These exceptions are not checked at compile time, so the programmer is responsible for handling these exceptions. Unchecked exceptions do not give compilation errors

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


int divide(int a, int b) {
    if (b == 0) {
        throw "Division by zero is not allowed.";
        //throw 123;
        //throw 5.001
    }
    return a / b;
}

int main() {
    try {
        int result = divide(10, 0);
        cout << "Result: " << result << endl;
    } catch (const char *ex) {
      cerr << "Exception caught: " << ex << endl;
        cout << "Exception caught: " << ex << endl;
    }
    catch (const int a){
      cerr << "Exception caught: " << a << endl;
        cout << "Exception caught: " << a << endl;
    }
    catch (...){
      cerr << "Exception caught: Error " << endl;
        cout << "Exception caught: Error " << endl;
    }

    return 0;
}
```

Virtual Function

- A virtual function is ***a member function in the base class that we expect to redefine in derived classes***.
- the most derived version of a class will be executed using a base class pointer

```
#include <bits/stdc++.h>
using namespace std;

class A{
  public:
    virtual void func(){
      cout << "A";
    }
};

class B:public A{
  public:
    void func(){
      cout << "B";
    }
};
int main() {
  A *p = new B();
  p->func();
}
```

- Pure Virtual Function
  - use the derived class function not base
  - every derived class must have the virtual function
  - Abstract class: which have at least one pure virtual function

  ```
  class A{
  ...
  virtual void func() = 0;
  ..
  }
  ```

OOP Characteristics

- **Encapsulation**
  - Wrapping of data and function together in a single unit known as Encapsulation
  - by default data are not accessible to outside of the class and they are accessible by member function which are wrapped in a class
  - prevention of data access by program → data hiding or information hiding
- **Data abstraction**

- Abstraction refers to the act of representing essential features without including the back ground details or explanation. Class use the concept of data abstraction so they are called **abstracted data type(ADT).**
- an abstract class must have at least one virtual function

```
class smartphone{
  ...
  public:
  virtual void camera() = 0;
  virtual void makeACall() = 0;
....
};

class iphone : public smartphone {
...
public:
void camera(){
...
//details or explanation
...
}
void makeACall(){
...
// details and explanation
...
}
};

class android : public smartphone{
...
public:
void camera(){
...
// details and explanation
...
}
void makeACall(){
...
// details and explanation
...
}
};

int main(){
  smartphone *a1 = new android();
  a1->camera();
  smartphone *a2 = new iphone();
  a2->makeACall();
/*
  The other developers just need to know that there's a function
named camera(), makeACall() ... So, we can hide the
implementation code of this.. the implementation code for each
child class may differ, but it's not a matter of consideration to
the other developers.
*/
}
```

- **Polymorphism**
  - *Greek* “Poly” (many) and “morphism” (forms) ⇒ many forms
  - it means the ability to take more than one form
  - Ways to achieve
    - Function overloading
    - Operator overloading
    - Function overriding
- **Inheritance**
  - Stated before

Function Overriding

- Function overriding in C++ is a concept by which you can define a function of the same name and the same function signature (parameters and their data types) in both the base class and derived class with a different function definition.
- It usually execute the derived class function then

```
class base {
...
public:
void func(){
  cout << "Base";
}
...
};

class derived : public base{
...
public:
void func() override{
  cout << "derived";
}
};

int main(){
  derived ob;
  ob.func(); // Output : Derived
}
```

```
// if we have to show the base class
class base {
...
public:
void func(){
  cout << "Base";
}
...
};

class derived : public base{
...
public:
void func() override{
  cout << "derived";
  // way 01:
  base::func();
}
};

int main(){
  derived ob;
  // Way : 02
  ob.Base::func(); // Output : Base
  // way : 03
  base ptr = new derived();
  ptr->func(); // Output : Base
}
```

- But there may be situations when a programmer makes a mistake while overriding that function. So, to keep track of such an error, C++11 has come up with the **override** identifier.

```
class base{
  ..
  public:
  void func(){
  ....
  }
};

class derived{
  ...
  public:
  void func(int a) override{
      .....
  }
};
```

```
int main(){
  derived a;
  a.func();
}
/*
Observation:
func() and func(int a) are not overriding, but overloading... by using "override" keyword,
it shows an error, because the function isn't overriding..
to stop making this mistake, override keyword is used.
*/
```

Friend Class

- Friend Class is a class that can access both private and protected variables of the class in which it is declared as a friend, just like a friend function.

  ```
  class One{
  .....
  friend class Two;
  };
  class Two{
  ........
  };
  ```

Question and Answers

- **Different between OOP and SPL**

Aspect	Object-Oriented Programming (OOP)	Procedural/Structured Programming
Paradigm	Based on the concept of objects and classes.	Based on procedures and functions.
Data and Behavior	Encapsulates data and behavior (methods/functions) together in classes.	Separates data and behavior (functions) into different entities.
Modularity	Emphasizes on creating reusable and modular code using classes and objects.	Relies on functions and procedures for code modularity.

Aspect	Object-Oriented Programming (OOP)	Procedural/Structured Programming
Abstraction	Supports data abstraction using access control (public, private, protected).	Limited or no direct support for data abstraction.
Inheritance	Supports inheritance, allowing one class to derive properties from another.	Typically does not support inheritance.
Polymorphism	Supports polymorphism, enabling one interface (function) to work with different data types.	Limited or no support for polymorphism.
Encapsulation	Encapsulation is a fundamental principle, where data and methods are bundled together within a class.	Encapsulation is not emphasized, and data may be freely accessed by different parts of the program.
Examples	C++, Java, Python, etc.	C, Pascal, Fortran, etc.
	It generally follows “Bottom-Up Approach”.	It generally follows “Top-Down Approach”.
	It gives more importance to data.	It gives more importance of code.

- **Here are some reasons why you might choose OOP for a complex project:**
  - **Modularity and Reusability:** OOP encourages breaking down complex problems into smaller, manageable modules (classes).
  - **Encapsulation:** OOP promotes data encapsulation, hiding the implementation details of classes and providing well-defined interfaces for interacting with objects
  - **Abstraction:** OOP allows you to abstract the essential characteristics of an object, focusing on what it does rather than how it does it. This level of abstraction simplifies understanding and makes it easier to manage complex systems.
  - **Inheritance:** OOP supports inheritance, which allows you to create new classes based on existing ones, inheriting their attributes and behaviors.
  - **Polymorphism:** OOP provides polymorphism, enabling you to use a single interface to represent different types of objects. This promotes flexibility and extensibility, making it easier to add new features or variations without modifying existing code.
  - **Encourages Design Patterns:** OOP aligns well with the use of design patterns, which are common solutions to recurring design problems.

    - **Team Collaboration:** OOP can promote better collaboration among team members. By dividing the project into classes and objects, different team members can work on different modules independently, leading to more parallel development and efficient teamwork.
    - **Code Maintainability:** OOP's modular and organized nature results in code that is easier to maintain and update
    - **Code Extensibility:** OOP makes it easier to extend functionality without modifying existing code.
    - **Readability and Understandability:** The structure and syntax of OOP are often more intuitive and natural, making the codebase easier to understand, especially when dealing with large and complex projects.
- **`cin`** and **`cout`** are function ? if not, then what are they ? how they works?
    - `cin` and `cout` are not functions; they are objects of the C++ Standard Library's `istream` and `ostream` classes, respectively. These objects are used for input and output operations in C++.

      Here's how they work:

      1. **`cin` (input stream object):**
          - `cin` is an object of the `istream` class, which is used for reading data from the standard input (usually the keyboard) into C++ variables.
          - It provides various **extraction operators** (`>>`) that allow you to read different types of data (e.g., integers, floating-point numbers, characters, strings) from the input stream.

         Example of reading an integer from the user:

         ```
         int num;
         std::cout << "Enter an integer: ";
         std::cin >> num; // User inputs an integer, and it is stored in 'num'
         ```

      2. **`cout` (output stream object):**
          - `cout` is an object of the `ostream` class, which is used for displaying data to the standard output (usually the console or terminal).
          - It provides various **insertion operators** (`<<`) that allow you to output different types of data to the output stream.

         Example of printing a message to the console:

```
std::cout << "Hello, World!";
```

Both `cin` and `cout` are part of the C++ Standard Library's `iostream` header, which provides facilities for input and output operations. They work with **overloaded operators** (`>>` for input and `<<` for output) to handle different data types and provide a convenient and concise way of interacting with the standard input and output streams.

When you use `cin` to read input, the program waits for the user to enter data and press the Enter key. The data is then read from the input stream, and the appropriate variables are populated with the entered values.

Similarly, when you use `cout` to output data, the data is sent to the output stream, and it appears on the console or terminal for the user to see.

Overall, `cin` and `cout` are essential tools for interacting with the user and providing informative output in C++ programs.

- **what are the roles of constructor and destructor in a class?**

  Constructor:

  - A constructor is a special member function with the same name as the class. It is automatically called when an object of the class is created.
  - The primary purpose of a constructor is to initialize the object's data members and set up its initial state.
  - Constructors are responsible for allocating resources, setting default values, and performing any necessary setup operations.
  - Constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists, allowing different ways to create objects.

  Destructor:

  - A destructor is a special member function with the same name as the class, preceded by a tilde `~`. It is automatically called when an object is about to be destroyed (e.g., when it goes out of scope or explicitly deleted).
  - The primary purpose of a destructor is to **release resources, perform cleanup, and deallocate memory** that was allocated during the object's lifetime.

    - Destructors are useful for managing dynamic memory, releasing file handles, closing network connections, or performing any cleanup tasks that the object requires before being removed from memory.
    - Unlike constructors, destructors **cannot be overloaded**. There can only be one destructor for a class.

- **Define the scope resolution operator?**
    - The scope resolution operator in C++ is denoted by `::` and is **used to access entities (variables, functions, classes, etc.) defined in different scopes**. It allows you to explicitly specify the scope of the entity you want to access, overriding the default scope resolution.
- **How do `new` and `delete` differ from `malloc` and `free` in c++ ?**
    - `new` and `delete` operators in C++ and `malloc()` and `free()` functions in C are used for dynamic memory allocation and deallocation, but they have some key differences:

    1. **Type Safety:**
        - `new` and `delete`: In C++, `new` and `delete` are type-safe. They automatically determine the size and type of the allocated memory based on the data type of the variable being allocated. The correct constructors and destructors are also called when using `new` and `delete` for objects, ensuring proper initialization and cleanup.
        - `malloc()` and `free()`: In C, `malloc()` and `free()` are not type-safe. They operate on `void*`, and you need to manually cast the returned pointer to the appropriate type. You also need to call the constructors and destructors explicitly for objects if needed.
    2. **Constructor and Destructor Calls:**
        - `new` and `delete`: They automatically call the constructor of the allocated object (if it's a class) during memory allocation and the destructor during deallocation. This ensures proper object initialization and cleanup.
        - `malloc()` and `free()`: They do not call constructors or destructors, making them suitable for raw memory allocation and deallocation without any additional initialization or cleanup.
    3. **Size Calculation:**

   - `new` and `delete`: They automatically calculate the size of the allocated memory based on the data type specified, so you don't need to explicitly mention the size during allocation or deallocation.
   - `malloc()` and `free()`: You must explicitly specify the size in bytes when using `malloc()` and pass the same size to `free()` during deallocation.
4. **Exception Handling:**
   - `new`: If memory allocation fails, it throws a `std::bad_alloc` exception, which can be caught and handled.
   - `malloc()`: If memory allocation fails, it returns a `NULL` pointer, and you need to check for this explicitly to handle memory allocation failures.
5. **Array Allocation:**
   - `new[]` and `delete[]`: In C++, you can allocate and deallocate arrays using `new[]` and `delete[]`. These operators keep track of the number of elements in the array for proper deallocation.
   - `malloc()` and `free()`: In C, `malloc()` can be used to allocate arrays, but it doesn't keep track of the number of elements. You need to manually store and manage the size of the allocated array.

- **List of types inheritance supported by C++.**

  In C++, there are four types of inheritance supported:

  1. **Single Inheritance:**
     - In single inheritance, a derived class inherits from only one base class.
     - It is the most common type of inheritance and represents a simple "is-a" relationship between classes.

     ```
     class Base {
         // Base class members
     };

     class Derived : public Base {
         // Derived class members
     };
     ```

  2. **Multiple Inheritance:**

- In multiple inheritance, a derived class can inherit from multiple base classes.
- It allows a class to combine features from multiple sources and is represented by a diamond-shaped inheritance diagram.

```
class Base1 {
    // Base class 1 members
};

class Base2 {
    // Base class 2 members
};

class Derived : public Base1, public Base2 {
    // Derived class members
};
```

3. **Multilevel Inheritance:**

- In multilevel inheritance, a derived class inherits from another class, which itself may be derived from another class.
- It forms a chain of inheritance relationships.

```
class Grandparent {
    // Grandparent class members
};

class Parent : public Grandparent {
    // Parent class members
};

class Child : public Parent {
    // Child class members
};
```

4. **Hierarchical Inheritance:**

- In hierarchical inheritance, multiple derived classes inherit from a single base class.
- It represents an "is-a" relationship, where different classes share common characteristics defined in the base class.

```
class Animal {
    // Base class members
};
```

```
class Dog : public Animal {
    // Dog class members
};

class Cat : public Animal {
    // Cat class members
};
```

- Hybrid Inheritance

- **What do you mean by generic programming? Write a program which will find**
  out the maximum from two numbers using template function.

Generic programming is a programming paradigm that aims to create reusable and flexible code by using templates or generics. In generic programming, algorithms and data structures are written in a way that they can work with different data types without having to rewrite the code for each specific type.

```
#include <iostream>

// Template function to find the maximum of two values
template <typename T>
T findMax(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    int intNum1 = 42, intNum2 = 73;
    double doubleNum1 = 3.14, doubleNum2 = 2.71;

    // Find the maximum of two integers
    int maxInt = findMax<int>(intNum1, intNum2);
    std::cout << "Maximum of " << intNum1 << " and " << intNum2 << " is " << maxInt << s
td::endl;

    // Find the maximum of two double values
    double maxDouble = findMax<double>(doubleNum1, doubleNum2);
    std::cout << "Maximum of " << doubleNum1 << " and " << doubleNum2 << " is " << maxDo
uble << std::endl;

    return 0;
}
```