Midterm Questions

1.What are the differences between references and pointers? Give an example.

1. Reference is an alias of the variable which it refer, whereas pointer is an object whose value is the address.
2. A pointer can be re-assigned while a reference cannot be re-seated after binding.
3. There is no reference arithmetics, such as ++. It is only the operator of the variable it bind.

```
int a = 1;
int & ref=a;
ref++;
cout<<a; //2
```

4. There is no constant reference

```
int & const ref=a //wrong!
```

5. No null reference, it must be initialized when defined.

2.What is operator overloading? What is a copy constructor? Write a class that includes operator= and a copy constructor.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
class MyBox{
    protected:
        int length;
        int width;
        int height;

    public:
        MyBox(int length=0,int width=0,int height=0):length(length),width(width),heig
ht(height){};

        void show_box(){
            cout<<"length:"<<length<<"width:"<<width<<"height:"<<height<<endl;
        }

        MyBox operator = (MyBox box2){
            cout<<" operator overload"<<endl;
            this->length = box2.length;
            this->height = box2.height;
            this->width = box2.width;
            return *this;
        }
    };
```

3. please explain to me what an automatic object is, and what its lifetime is. How to recover the pointer locally not globally?

As object is a kind of variable. Say, the automatic variable is allocated and deallocated automatically when program flow enter and leaves the variable's scope.

Pointer is the same as it is a kind of variable. When we use delete operator to recover the pointer, it means the pointer is recovered locally. Wherease we don't do anything, when the process is at the expiration of the scope which contains the pointer(e.g. for-loop or main function), say, the variable(pointer) defined within it would be recovered globally.

4. How do virtual functions work in C++? Give an example

virtual function is the function that is later redefined in each of the derived class. Moreover, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class that is pointing to an object of the derived class.

```
class AbstractCar{
protected:
int price;
public:
    AbstractCar(int price):price(price){};
    virtual void show_price()=0;
};

class Porsche: AbstractCar{
public:
    Porsche(int price):AbstractCar(price){};
    void show_price(){
        cout<<"the porsche's price is:"<<this->price<<endl;
    }
};
```

5. What difference is between overriding and overloading? Give an example

- Overloading generally means that you have two or more functions in the same scope having the same name.

For example:

```
int add(int a,int b){
    return a+b;
}
double add(double a, double b){
    return a+b;
}
```

- Overriding is a different concept that doesn't compete with overloading. When you have a virtual function in base class, you can write a function with the same signature as in virtual function in your derived class. This means the derived class's method override the base class's method.

For example:

```
class AbstractCar{
protected:
    int price;
public:
    AbstractCar(int price):price(price){};
    virtual void show_price()=0;
};

class Porsche: AbstractCar{
public:
    Porsche(int price):AbstractCar(price){};
    void show_price(){
        cout<<"the porsche's price is:"<<this->price<<endl;
    }
};
```

6. What is a virtual destructor? When do you decide to use virtual destructor instead of regular destructor?

Virtual destructor is the destructor of class has a key word virtual. It avoid a situation that when a pointer to the base class point to a derived class and then, delete the pointer can result in an undefined behavior when destructor of base class is non-virtual.

It is recommended that when you have a virtual function in class(base class), you should add a virtual destructor immediately.

For example:

```
class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
```

7. how many ways we can initialize an int variable in C++? Also, give the examples

- c-like: `int a =1;`
- constructor initialization: `int b(2);`
- uniform initialization: `int c={1};`

8. what is the implicit conversion in C++?.what do you mean by internal linking and external linking in C++?

Implicit conversions are performed whenever an expression of some type T1 is used in context that does not accept that type, but accepts some other type T2; in particular:

1.

- when the expression is used as the argument when calling a function that is declared with T2 as parameter;
- when the expression is used as an operand with an operator that expects T2;
- when initializing a new object of type T2, including return statement in a function returning T2;
- when the expression is used in a switch statement (T2 is integral type);
- when the expression is used in an if statement or a loop (T2 is bool).

2. external linkage means the symbol (function or global variable) is accessible throughout your program and internal linkage means that it's only accessible in one translation unit

For example

explicitly control the linkage of a symbol by using the extern and static keywords.

```
static int a//internal
external int b//external
namespace{
    int i; //external but unreachable from other translation units.
}
```

9. Is it possible to have a recursive inline function? When you use inline? When you use recursive function?

It depends on compiler. But roughly speaking, it's not recommended to do so. As inline recursive function may cause infinitely unroll the function.

We use inline function to save time of calling function. And inline functions are always simple.

We use recursive function to solve recursive questions.

10. What is the difference between a template class and class template?

- Template Class

It is unpreciesly to say template class, as it is always a misreading of "template for class" as opposed to a "function template," which is "a template for a function. Class do not define templates!

- Class Template

A class template defines a family of classes.

For example:

```
template<typename T>
class Array {...};
```

But it is a template, not a class. The declarition Array is a class or pedantically, a class based on a template.

11. Explain what is Polymorphism in C++? Give an example

Polymorphism is the provision of a single interface to entities of different types.

For example:

```
class CarFactory{
public:
    CarFactory(string factoryType):factoryType(factoryType){};
    virtual AbstractCar* createCar(int price=0)=0;
protected:
    string factoryType;
};

class PorscheFactory : public CarFactory{
public:
    PorscheFactory():CarFactory("Porsche"){};
    AbstractCar* createCar(int price=0){
        return new Porsche(price);
    }
};

class LamborghiniFactory : public CarFactory{
public:
    LamborghiniFactory():CarFactory("Lamborghini"){};
    AbstractCar* createCar(int price=0){
        return new Lamborghini(price);
    }
};
```

12. How can I exchange char array into a string? How can I assign a char point with an existed char type?

```
string my_str="hello world";
char char_array[10]="how r you";
```

- exchange char array into a string

  ```
  my_str=char_array;
  cout<<my_str<<endl;
  ```

- assign a char pointer with an exist char type

```
char * char_ptr;
char my_char= 'c';
char_ptr=char_array;//char array
char_ptr= &my_char;//char type
```

13. What is the difference between variable declaration and variable definition? Give an example

- A declaration provides basic attributes of a symbol: its type and its name. Declaration does not provide the body of the function/variable/class, but it does tell the compiler that it can use this function and expect that it will be defined somewhere.

  For example:

  ```
  int myfun(int a);//declaration
  int x;//both declaration and definition
  external int x;//just declaration
  ```

- A definition provides all of the details of that symbol--if it's a function, what it does; if it's a class, what fields and methods it has; if it's a variable, where that variable is stored. Defining a function means providing a function body For example:

  ```
  int x;//both declaration and definition
  int myfun(int a){
      return a;
  }//definition
  ```

- Particularly, as for variable, when you declare a variable, at most of time, you are also providing the definition. `int x;` What does it mean to define a variable, exactly? It means you are telling the compiler where to create the storage for that variable. However, `external int x;` is creating a declaration of a variable but NOT defining it; it is saying that the storage for the variable is somewhere else.

14. How many types of inheritance supported in C++. Give an example of each type.

When derived a class from a base class. There are three types of inheritance: public, protected and private(access-specifier).

1. Public Inheritance: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected

members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

2. Protected Inheritance: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

3. Private Inheritance: When deriving from a private base class, public and protected members of the base class become private members of the derived class.

In general, there are five different types of inheritance supported in C++:

1. Single inheritance: One class is delivered from one base class.

```
class Sportscar{
protected:
    int speed=180;
};

class Porsche: public Sportscar{
public:
    void show_speed(){
        cout<<speed<<endl;
    }
};
```

2. Multiple inheritance: One class is delivered from multiple base class.

```
class Sportscar{
protected:
    int speed=180;
public:
    void show_speed(){
        cout<<"this car's speed is "<<speed<<endl;
    }
};
class Muslecar{
protected:
    int strength=500;
public:
    void show_strength(){
        cout<<"this car's strength is "<<strength<<endl;
    }
};
class Porsche: public Sportscar{};
class Mustang: public Sportscar, public Muslecar{};//multiple inheritance
```

3. Hierarchical Inheritance: Multiple classes are delivered from one base class.

```
class Sportscar{
protected:
    int speed=180;
public:
    void show_speed(){
        cout<<"this car's speed is "<<speed<<endl;
    }
};

class Porsche: public Sportscar{};
class Lamborghini: public Sportscar{};
```

4. Multilevel Inheritance: The derived class inherits from a class which in turns inherits from some other class.

```
class Sportscar{
protected:
    int speed=180;
public:
    void show_speed(){
        cout<<"this car's speed is "<<speed<<endl;
    }
};

class Porsche: public Sportscar{};

class Porsche911 : public Porsche{
protected:
    int value= 1000000;
};
```

5. Hybrid Inheritance(Virtual Inheritance) The combination of hierachical inheritance and multiple inheritance.

```
class B{ public:int b=1;};
class B1:  virtual  public B{ private:int b1;};
class B2:  virtual  public B{ private:int b2;};
class C: public B2, public B1{};
int main() {
    C c;
    cout<< c.b << endl;
}
Result:
1
```

15: What is the real purpose of class? What things would you remember while making an interface?

Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.

Class is a abstraction of the real world. The concepts of class is emerging with object oriented programming.

Programming in classes has a lot of advantages such as, code encapsulation, modularization, reusability and etc.

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class. The C++ interfaces are implemented using abstract classes. A class is made

abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
    class AbstractCar{
    protected:
        string car_type;
    public:
        AbstractCar(string carType):car_type(carType){};
        virtual void show_type()=0;
    };
```

16: Does loop has a switch will be better than use two loops?

It all depends! There is no way to conclude that loop is better than switch or switch is better than loop.

Cause loop and switch are two different abstraction of logic.

For example when we deal with matrix printing, we use two loop rather than one loop with a switch:

```
for(int i=1;i<5;i++){
   for(int j=1;j<5;j++){
      cout<<"["<<i<<","<<j<<"]";
    }
    cout<<endl;
}
```

On the other hand, when we come up with a classification problem, such as voting, the switch take place. For example:

```
int count1;
int count2;
for(int i=0;i<10;i++){
    int party;
    cout<<"please input which party do you like to support: (1/2)"<<endl;
    cin>>party;
    switch (party) {
        case 1:
            count1++;
            break;
        case 2:
            count2++;
            break;
        default:
            cout<<"invalid vote!"<<endl;
            break;
    }
}
cout<<"party1:"<<count1<<"\n"<<"party2:"<<count2<<endl;
```

17: What is conversion constructor? Write an example.

A constructor that is not declared with the specifier explicit and which can be called with a single parameter (until C++11) is called a converting constructor.

```
struct A{
protected:
    int value;
    int data;
public:
    A(){};//converting constructor (since c++11)
    A(int a):value(a){};//converting constructor
    explicit A(int a, int b):value(a),data(b){};//  explicit constructor
};
int main(){
    A aa(10);//OK: direct-initialization selects A::A(int)
    A a1=10;//OK: copy-initialization selects A::A(int)
    A a2={10};//OK: copy-list-initialization selects A::A(int)
    //A a3={1,2}; //error: copy-list-initialization selected an explicit constructor
```

18: Are the exceptions and error the same?

Briefly speaking, exceptions are those which can be handled at the run time whereas errors cannot be handled.

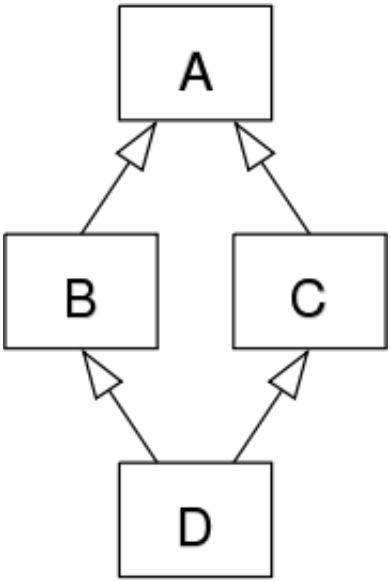
In c++,exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.

To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

For example:

```
try{
    throw 10;
}
catch(int e){
    cout<<"the exception number: "<<e<<endl;
}
```

19: What is the "diamond problem" that can occur with multiple inheritances? Give an example.



Suppose we have 2 classes B and C that derive from the same class – in our example above it would be class A. We also have class D that derives from both B and C by using multiple inheritance. You can see in the figure above that the classes essentially form the shape of a diamond – which is why this problem is called the diamond problem.

```
class Animal{
    int weight;
public:
    int getWeight() { return weight;};
};
class WaterAnimal : public Animal {};
class LandAnimal : public Animal {};
class Amphibian : public WaterAnimal, public LandAnimal {};

int main(){
    Amphibian frog;
    frog.getWeight();//wrong!
```

Result:

In our inheritance hierarchy, we can see that both the WaterAnimal and LandAnimal classes derive from the Animal base class. And here is the problem: because Amphibian derives from both the WaterAnimal and LandAnimal classes – which each have their own copy of the data members and methods of the Animal class- the Amphibian object "frog" will contain two subobjects of the Animal base class.

Solution to the Diamond Problem:

Virtual Inheritance. If the inheritance from the Animal class to both the WaterAnimal class and the LandAnimal class is marked as virtual, then C++ will ensure that only one subobject of the Animal class will be created for every Amphibian object.

```
class Animal{
    int weight;
public:
    int getWeight() { return weight;};
};
class WaterAnimal : virtual public Animal {};
class LandAnimal : virtual public Animal {};
class Amphibian : public WaterAnimal, public LandAnimal {};
```

20: What will happen if I use delete, but actually I should use delete[]? What will happen if you delete a pointer twice? How to fix it if it is a problem?

1. What will happen if I use delete, but actually I should use delete[]?

   Generally, delete is corresponding to new whereas delete[] is corresponding to new[]. Misusing the delete operators always throw an error.

Firstly, consider the class(object) situation:

```
Sportscar* sportscar =new Sportscar;
delete sportscar;
//delete[] Sportscar;//wrong!
Sportscar* sportscarArray=new Sportscar[3];
sportscarArray[0]=Sportscar("666666");
sportscarArray[1]=Sportscar("999999");
sportscarArray[2]=Sportscar("A12345");
for(int i=0; i<3;i++){
    cout<<"the car number of "<<i<<" is "<<sportscarArray[i].get_car_number()<<en
dl;
}
delete[] sportscarArray;
//delete sportscarArray;//wrong!
```

Conclusion: If we use wrong operator corresponding to the pointer initialization. It always throw an error such like:

Q20(53161,0x7fff7d240300) malloc: *** error for object 0x1001000f8: pointer being freed was not allocated *** set a breakpoint in malloc*error*break to debug

Secondly, considering the base type:

```
int * int1 = new int(1);
delete[] int1;//ok
//delete int1;
int * int2 = new int[3];
int2[0]=1;
int2[1]=2;
int2[2]=3;
for(int i=0;i<3;i++){
    cout<<"the int array["<<i<<"] is "<<int2[i]<<endl;
}
delete int2;//ok
//delete[] int2;
```

Conclusion: If the data type belongs to basic data type, such as int, long, char and etc. delete can be used to replace delete[]

2. What will happen if you delete a pointer twice? How to fix it if it is a problem?

   Let's delete a pointer twice:

```
delete int2;//ok
//delete[] int2;
delete int2;
```

It throw out an error such like:

Q20(53300,0x7fff7d240300) malloc: *** error for object 0x1002046b0: pointer being freed was not allocated *** set a breakpoint in malloc*error*break to debug

Solution: For each time you delete a pointer, you should assign the pointer to a null pointer(in c++, use nullptr instead). That would avoid dangling pointers.

```
delete int2;
int2 = nullptr;
delete int2;
int2 = nullptr;
```