(1) Do use use CVS, SVN ?
Yes, cvs co, cvs commit, cvs add, cvs diff

(2) Do you use IDE for coding and programming ?
Never heard about it
An integrated development environment (**IDE**) or interactive development environment is a software application that provides comprehensive facilities to computer programmers for software development. An **IDE** normally consists of a source code editor, build automation tools and a debugger.

C/C++ basic concepts:

1. Basic knowledge of C/C++

1 C++ is **Multi-Paradigm** (not pure OOP, supports both procedural and object oriented) while C follows procedural style programming.

目前主流的编程范式有：命令式编程(Imperative programming)、函数式编程(Functional programming)、面向对象编程(Object-oriented programming)等。我们普通码农最熟悉的应该就是面向对象编程了。

2 The data in C has less security, however, in C++ you can define your class members with type of **public, private, protected.** This will make your data accessible at different security level.

3 C follows top-down approach (solution is created in step by step manner, like each step is processed into details as we proceed) but C++ follows a bottom-up approach (where base elements are established first and are linked to make complex solutions).

4 **C++ supports function overloading while C does not support it**. For example, in the same class, you can define function with the same name, but different type of arguments or parameters.

5 C++ allows use of functions in structures, but C does not permit that. Structure in C++ behaves like class, it allows methods, constructors, destructor in C++, but not in C.

6 **C++ supports reference variables** (two variables can point to same memory location). C does not support this.

7 C does not have a built in exception handling framework, though we can emulate it with other mechanism. **C++ directly supports exception handling**, which makes life of developer easy.

2. What is a class?

A class is user defined datatype or data structure. It has data and functions as its members. The access of the data and members is governed by three access specifiers: public private, and protected.

3. What is an Object/Instance?

Object is the instance of a class, which is concrete. After you defined the class, you can create an object using the constructor function in that class.
For example, the definition Country could be a class, you can define the data member population, territory. A particular country for example Unite State is a object of the class.

4. What do you mean by C++ access specifiers ?
The default access level is private.
public:
Members declared as public can be accessed from any where. As long as you include the class header file, define a pointer or object, you can get the data wherever you want.
private:
Members declared as private **are accessible only with in the same class** and cannot be accessed outside the class even the derived class.
protected:
Members declared as protected can not be accessed from outside the class **except a child class**. This access specifier has significance in the context of inheritance.

5. Whats difference between struct in C and class in C++?
1st you can not specify the data type as “public, private, and protected” in the struct in C. Members of a struct in C is public by default, however, in C++ you can specify the accessable level of the data member. This will enhance the data security.
2n struct in C does not support function, constructor, destructor that used in class.

6. What is the basic concepts of Object-Oriented-Programming (OPP)
• **Classes and Objects**
• **Encapsulation**
Data encapsulation is a mechanism of bundling the data, and the functions together that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation can be achieved by using the function access specifiers (private, and protected).

```
1   class Country
2   {
3     private:
4       int population;
5     public:
6       int getPopupation(){
7         return population;
8       }
9       int setPopulation(int value){
10        if(value > 0){
11            population = value;
12        }
13      }
14  };
```

• **Data abstraction**

Data abstraction refers to hiding the background implementations and only show the necessary details to the outside world. Data abstraction is implemented using **interfaces** and **abstract classes**.
We define the base class as “abstract class” for **interfaces** using **pure virtual function** [virtual int pool()=0]

```
1   class Stack
2   {
3       public:
4           virtual void push(int)=0;
5           virtual int pop()=0;
6   };
7
8   class MyStack : public Stack
9   {
10      private:
11          int arrayToHoldData[]; //Holds the data from stack
12
13      public:
14          void push(int) {
15              // implement push operation using array
16          }
17          int pop(){
18              // implement pop operation using array
19          }
20  };
```

Then implement a derived class to implement the details.

• **Inheritance**
Inheritance allows one class to inherit properties of another class. In other words, inheritance allows one class to be defined in terms of another class. For example, you can define a class called “AisaCountry” which inherit from the class “country”. We can access the data members, methods of the mother class via the child object or pointer.

```
1   class SymmetricShape
2   {
3       public:
4           int getSize()
5           {
6               return size;
7           }
8           void setSize(int w)
9           {
10              size = w;
11          }
12      protected:
13          int size;
14  };
15
16  // Derived class
17  class Square: public SymmetricShape
18  {
19      public:
20          int getArea()
```

```
21          {
22              return (size * size);
23          }
24    };
```

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Type of Inheritance:
When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.
We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

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

```
public > protected > private
```

7. Example on polymorphism, overriding, overloading.
Polymorphism occurs when you have class inheritance. If you define two functions with the same name and same argument, while the one in base class is virtual. With the pointer of the base class, a call to the member function will invoke the function in the derived class to be executed.
Overriding means that if the **base class and derived class** have function with the **same name same arguments**. With the pointer and object of the derived class, the function in the derived class will be involved. ie. function in the derived class overrides the function in the base class.
Overloading means you can define functions with the same name but different arguments in the same class.

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

class A {
        public:
        virtual void func() {cout << "A" <<endl;};
        void overd() {cout<<" overrid A "<<endl;}
};
```

```
class B: public A {
        public:
        void func() {cout << "B" <<endl;}
        void overd() {cout<<" overrid B "<<endl;}
        void overd(int a) {cout<<" overrid B: "<< a <<endl;}
};
int main(){
A *a=new A();
B *b=new B();
a->func();    /// func() in A will be involked
b->func();    /// func() in B will be involked
a->overd();   /// overd() in A will be called
b->overd();  ///   function overriding, overd() in class B will be called
b->overd(5);  /// function overloading

 A *a1 = new B();
 a1->func();   //// polymophism when using "virtual", func() in class B will be
involked
a1->overd();  /// overd() in class A will be called
// a1->overd(7); /// can not do this, will give error since overd(int a) is not
defined in A
return 0;
}
```

8) why is virtual class used?

An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.

C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

For example you have class A, both class B and C inherit from A. When you define a class D inherit from B and C, now you will have trouble with duplicate inheritance problem. While if you define B, and C as virtual inheritance. The duplicate problem will be solved.

```
class A
{
    public:
        int i;
};
class B : virtual public A
{
     public:
           int j;
};
class C: virtual public A
{
      public:
          int k;
};
class D: public B, public C
```

```
{
public:
     int sum;
};
int main()
{
        D ob;
        ob.i = 10; //unambiguous since only one copy of i is inherited.
        ob.j = 20;
        ob.k = 30;
        ob.sum = ob.i + ob.j + ob.k;
        cout << “Value of i is : ”<< ob.i<<”\n”;
        cout << “Value of j is : ”<< ob.j<<”\n”; cout << “Value of k is :”<< ob.k<<”\n”;
        cout << “Sum is : ”<< ob.sum <<”\n”;

return 0;
}.
```

9) why declare a class as 'friend'?

As we said before, private and protected can not be accessed from outside of the class. However, with friend class defined, you can access the private and protected members of a class due to friendship.

```
// friend class
#include <iostream>
using namespace std;

class Square;

class Rectangle {
   int width, height;
  public:
   int area ()
     {return (width * height);}
   void convert (Square a);
};
class Square {
  friend class Rectangle;
  private:
   int side;
  public:
   Square (int a) : side(a) {}
};
void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}
int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

10. In how many ways we can initialize an int variable in C++?
In c++, variables can be initialized in two ways, the traditional C++ initialization using "=" operator and second using the constructor notation.
int i =10;
int i(10);

11. What is implicit conversion/coercion in c++?
Implicit conversions are performed when a type (say T) is used in a context where a compatible type (Say F) is expected so that the type T will be promoted to type F.
int a = 2 ;
float b = a+3. ;

12. What are C++ inline functions?
C++ inline functions are special functions, for which the compiler replaces the function call with body/definition of function. Inline functions makes the program execute faster than the normal functions, since the overhead involved in saving current state to stack on the function call is avoided. By giving developer the control of making a function as inline, he can further optimize the code based on application logic. But actually, it's the compiler that decides whether to make a function inline or not regardless of it's declaration. Compiler may choose to make a non inline function inline and vice versa. Declaring a function as inline is in effect a request to the compiler to make it inline, which compiler may ignore. So, please note this point for the interview that, it is upto the compiler to make a function inline or not.

13. What do you mean by translation unit in c++?
We organize our C++ programs into different source files (.cpp, .cxx etc). When you consider a source file, at the preprocessing stage, some extra content may get added to the source code (for example, the contents of header files included) and some content may get removed (for example, the part of the code in the #ifdef of #ifndef block which resolve to false/0 based on the symbols defined). This effective content is called a translation unit. In other words, a translation unit consists of

- Contents of source file
- Plus contents of files included directly or indirectly
- Minus source code lines ignored by any conditional pre processing directives (the lines ignored by #ifdef,#ifndef etc)

14） How many storage classes are available in C++?
Every C variable has a storage class and a scope. The storage class determines the part of memory where storage is allocated for an object and how long the storage allocation continues to exist. It also determines the scope which specifies the part of the program over which a variable name is visible, i.e. the variable is accessible by name. The are four storage classes in C are automatic, register, external, and static.

15） What are virtual functions and what is its use?
Virtual functions are member functions of class which is declared using keyword 'virtual'. When a base class type reference is initialized using object of sub class type and an overridden method which is declared as virtual is invoked using the base reference, the method in child class object will get invoked.

```
1   class Base
2   {
3       int a;
4       public:
5           Base()
6           {
7               a = 1;
8           }
9           virtual void method()
10          {
11              cout << a;
12          }
13  };
14
15  class Child: public Base
16  {
17      int b;
18      public:
19          Child()
20          {
21              b = 2;
22          }
23          virtual void method()
24          {
25              cout << b;
26          }
27  };
28
29  int main()
30  {
31      Base *pBase;
32      Child oChild;
33      pBase = &oChild;
34      pBase->method();
35      return 0;
36  }
```

22. What is virtual destructors? Why they are used?
Virtual destructors are used for the same purpose as virtual functions. When you remove an object of subclass, which is referenced by a parent class pointer, only destructor of base class will get executed. But if the destructor is defined using virtual keyword, both the destructors [of parent and sub class] will get invoked.

23）What do you mean by pure virtual functions in C++? Give an example?
Pure virtual function is a function which doesn't have an implementation and the same needs to be implemented by the the next immediate non-abstract class. (A class will become an abstract class if there is at-least a single pure virtual function and thus pure virtual functions are used to create interfaces in c++).

How to create a pure virtual function?

A function is made as pure virtual function by the using a specific signature, **" = *0*"** appended to the function declaration as given below,

```
class SymmetricShape {
    public:
        // draw() is a pure virtual function.
        virtual void draw() = 0;
};
```

24） What is function overloading and operator overloading?

Function overloading: C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

Operator overloading allows existing C++ operators to be redefined so that they work on objects of user-defined classes. Overloaded operators are syntactic sugar for equivalent function calls. They form a pleasant facade that doesn't add anything fundamental to the language (but they can improve understandability and reduce maintenance costs).

25） How compile and link works for C++ ?

1. Preprocessing: the preprocessor takes a C++ source code file and deals with the #includes, #defines and other preprocessor directives. The output of this step is a "pure" C++ file without pre-processor directives.
2. Compilation: the compiler takes the pre-processor's output and produces an object file from it.
3. Linking: the linker takes the object files produced by the compiler and produces either a library or an executable file.

26) What will happen if I say delete this

The destructor will be executed, but memory will not be freed (other than the work done by destructor).

27) What is the difference between overloading and overridding?

Overloading - Two functions having same name and return type, but with different type and/or number of arguments.

Overriding - When a function of base class is re-defined in the derived class.

28) Explain the need for Virtual Destructor.

Destructors are declared as virtual because if do not declare it as virtual the base class destructor will be called before the derived class destructor and that will lead to memory leak because derived classs objects will not get freed.Destructors are declared virtual so as to bind objects to the methods at runtime so that appropriate destructor is called.

29）What is storage class, how many storage class.
A storage class defines the scope (visibility) and life-time of variables, object, or functions within a C++ Program. There are 4 storage classes:

- auto, register, static, extern

The **auto** storage class is the default storage class for all local variables. auto automatically define the data type.

```
{
  int mount;
  auto int month;
}

{
auto x = 2.2;   ===== auto double x=2.2;  ///// auto think x is “double”
}
for(auto it=myArr.begin(); it != myArr.end(); ++it) cout<<"  "<< *it;
for(array<int, 6>::iterator iter=myArr.begin();  iter!= myArr.end(); ++iter) cout<<"  "<< *iter;
```

here auto think “it=array<int, 6>::iterator”

The **register** storage class is used to define local variables that should be stored in a register (寄存器) instead of RAM（内存）. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  miles;
}
```

static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;
int Road;
      {
         printf("%d\n", Road);
      }
```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in. static can also be defined within a function. If this is done the variable is initalised at run time but is not reinitalized when the function is called. This inside a function static variable retains its value during vairous calls.
static variable only have internal linkage.

extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initalized as all it does is point the variable name at a storage location that has been previously defined.
When you have multiple files and you define a global variable or function which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to decalre a global variable or function in another files.
default is always external linkage in C, while in C++ it's internal for constant variables.
File 1: main.c

```
  int count=5;
  main()
```

```
  {
    write_extern();
  }
File 2: write.c
  void write_extern(void);

  extern int count;

  void write_extern(void)
  {
    printf("count is %i\n", count);
  }
```

29）What is the difference of HD, CPU register, memory.
Register is on the physical processor uint, and it is faster than memory.
HD is the slowest one compare register and memory.

30）Why use const variable ? why const object ? why constant function ?
If a variable is defined a const type, it can not be modified.
If a object of a class is defined as const, the “data member, and member functions” can bot be changed. const class object can only call “const member functions”
A “const member function” is a member function that guarantees it will not change any class variables or call any non-const member functions.

```
#include "iostream"
using namespace std;

class Something {
    public:
        int m_nValue;

        Something() { m_nValue = 0; }

        void ResetValue() { m_nValue = 0; }
        void SetValue(int nValue) const { m_nValue = nValue; }    ////// wrong, const func can not change member

        int GetValue() { return m_nValue; }
};
int main() {
    const Something cSomething; // calls default constructor
    cSomething.m_nValue = 5; // violates const
    cSomething.ResetValue(); // violates const
    cSomething.SetValue(5); // violates const

    return 0;
}
```

The right one is:
```
#include "iostream"
```

```
using namespace std;

class Something {
   public:
      int m_nValue;

      Something(int a) { m_nValue = a; }
            void ResetValue() { m_nValue = -9999.; }
      void SetValue(int a) { m_nValue = a; }
      int GetValue() const { return m_nValue; }
      int GetValue() { return m_nValue; }
};

int main()
{
   const Something cSomething(9); // calls default constructor
   Something cSomething1(99); // calls default constructor
 cSomething1.ResetValue();
   cSomething1.SetValue(999);

   cout<< "Const one: " << cSomething.GetValue() <<"\t Normal:
"<<cSomething1.GetValue()<<endl;

   return 0;
}
```

31）Why use reference ? Pointer vs. Reference.
A reference variable is an alias, that is, another name for an already existing variable.

```
int    i = 17;
int&   r = i;
int *a = &i
*a = 17
a=0x**
```

30）Date and time.
ctime from c

31) What is the difference between realloc() and free()
free() frees a block of memory previous allocated using malloc()
realloc() changes the size of the block of memory pointed by a pointer.
memset()
malloc()

```
#include <stdlib.h>     /* malloc, calloc, realloc, free */

int main (){
  int * buffer1, * buffer2, * buffer3;
  buffer1 = (int*) malloc (100*sizeof(int));
  buffer2 = (int*) calloc (100,sizeof(int));
  buffer3 = (int*) realloc (buffer2,500*sizeof(int));
  free (buffer1);
```

```
  free (buffer3);
  return 0;
}
```

32) What is the advantage of using inheritance
Code reusable.

33) What is virtual destructors / constructors ?
Virtual destructors: When you delete a pointer of the base class, which point to a object of the derived class. Both destructors in derived class and base class will be involked.
Virtual constructor is not allowed. The virtual keyword is used when we want to introduce the polymorphic behavior, however, the constructor do not have polymorphism.

```
#include iostream.h
class Base{
      public:
       Base(){ cout<<"Constructing Base";}
      // this is a destructor:
      virtual ~Base(){ cout<<"Destroying Base";}
};
class Derive: public Base
{
        public:
           Derive(){ cout<<"Constructing Derive";}
           ~Derive(){ cout<<"Destroying Derive";}
 };

void main()
{
      Base *basePtr = new Derive();
     delete basePtr;
}
Output is:
Constructing Base
Constructing Derive
Destroying Derive
Destroying Base
```

34) What is difference between array, vector and list ?
For Array vector, is used to store the same type data variable. you can access via a[i]
For list, you need to iterate with a loop

35) What is a template ?
A template is defined to allow generic functions that admit any data type as parameters and return values without having to overload the function. template <>
vector <int> vec1

36) explain difference between new() and malloc()
new delete and preprocessor, malloc() free() are functions.
no need to allocated memory for new. malloc() use the sizeof() to allocate memory size.

37) What is friend function ?
As a friend of a class, the class can access the private and protected member of it.
A friend function is not a member if the class, but must be listed in the definition. It can directly access the member of the class.

```
#include <iostream>
using namespace std;
class Box{
   double width;
public:
   friend void printWidth( Box box );
   void setWidth( double wid );
};
// Member function definition
void Box::setWidth( double wid ) {
    width = wid;
}
// Note: printWidth() is not a member function of any class.
void printWidth( Box box ) {
   /* Because printWidth() is a friend of Box, it can
    directly access any member of this class */
   cout << "Width of box : " << box.width <<endl;
}
```

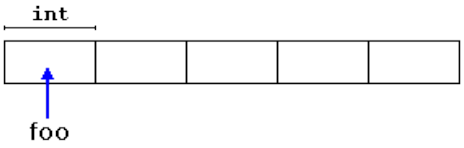
38) Dynamic memory in C, malloc, calloc, realloc and free
Dynamic memory in C++ operator new, new[], delete, delete[] in header file <new>

```
int main (){
 MyClass * p1 = new MyClass;
 // allocates memory by calling: operator new (sizeof(MyClass))
 // and then constructs an object at the newly allocated space

 MyClass * p2 = (MyClass*) ::operator new (sizeof(MyClass));
 // allocates memory by calling: operator new (sizeof(MyClass))
 // but does not call MyClass's constructor
 delete p1;
 delete p2;

 // allocates and constructs five objects:
 MyClass *p3 = new MyClass[5];
 delete [] p3;

 return 0;
}
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type `type`. The second one is used to allocate a block (an array) of elements of type `type`, where `number_of_elements` is an integer value representing the amount of these. For example:

```
int * foo;
foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type `int` and returns a pointer to the first element of the sequence, which is assigned to `foo` (a pointer). Therefore, `foo` now points to a valid block of memory with space for five elements of type `int`.



39) What wrong with code ?
T *p =0;
delete p;
It will crash since you delete a NULL pointer.

40) What wrong with code :
T *p = new T[10];
delete p;
It only delete the first element of the array.

41) What does extern function mean ?
Mean this function can be used outside the file.

42) difference between char a[]=“sss”, char *p = “sss”;
for a[], a[0]=“s”, 3 bytes allocated in a.
for p, p is a pointer point to char.

43)
1.) int (*p)[5]; //// int *p[5]
p[0] = new int(4), p[0] to p[4] is the pointer of int, p is the pointer point to the int pointer array
2.) char *x[3][4];
char *p=“sss”, char *p[2]={xxx, sss}, char *p[2][2]={“s”,”d”,”f”,”g”}
so, x is array of 3x4 pointers to char
3.) int *(*a[10])();
int a() is a function return int; int* a() is a func return int pointer.
int* (*a)() a pointer of fun return int pointer.
int* *a[10]() a pointer of fun return int pointer.
4.) int (*t[])();
a array of pointer return int

44) Difference between stack and queue,
Both are container adaptor
stack is Last-In First-Out (LIFO), queue is First-In First-Out (FIFO).
Both support empty, size, front, back, push_back, pop_front operations.

45) Tell how to check whether a linked list is circular.
A: Create two pointers, each set to the start of the list. Update each as follows:

```
while (pointer1) {
 pointer1 = pointer1->next;
 pointer2 = pointer2->next; if (pointer2) pointer2=pointer2->next;
 if (pointer1 == pointer2) {
   print (\"circular\n\");
 }
}
```

46) Macro (宏) in C and inline function in C++ ?

```
#define S(a, b) a*b
inline float A::S(float a, float b) {return a*b}
```

Macros are declarations that are substituted by the preprocessor (before the actual *compile*) The are not functions at all. It will be expanded in the compilation process.
An inline function declared as a normal function but defined outside the class using keyword inline.
Inline functions are a lot like a placeholder. Once you define an **inline function**, using the '**inline**' keyword, whenever you call that **function** the compiler will replace the **function** call with the actual code from the **function**.

```
inline float StHltEvent::innerSecGain() const {return mInnerSecGain;}
```

47) what #define sq(x) x*x
In my understanding, macro is processed in the preprocessing in compilation.
sq(a+b) = a+b*a+b
should #define sq(x) ((x)*(x))

48) What is constructor or ctor?
Constructor creates an object and initializes it. It also creates vtable for virtual functions. It is different from other methods in a class.
49) Scope resolution :: operator is used to resolve for the global scope of a variable if the local and global variable conflict by name.
50) What is the output of the following program ?

```
#include<isotream>
using namespace std;
void f() {
   static int i;
   ++i;
   cout<<i<<" ";
   }
main(){
   f();
   f();
   f();
```

```
}
The output is 1 2 3, since A static local variables retains its
value between the function calls and the default value is 0.
You should use auto, should be 1 1 1
```

49) What is bitwise ? How to use the NOT(~), AND (&), OR (|), XOR (^)
NOT, ~ : int a = 2, int b = ~a, b = -3
a = 0000 0000 0000 0010, ~a= 1111 1111 1111 1101 (-3 补码),
AND &,
OR |
XOR ^: int a = 2, int b=5
a^b = 0010 ^ 0101 = 0111 (only true and two bit is different)

```
#include<isotream>
using namespace std;
main()
{
   int i = 13, j = 60;

   i^=j;
   j^=i;
   i^=j;
   cout<<i<<” “<<j;
}
```

50) What is the output of the following program?

```
#include<isotream>
using namespace std;
class Base {
public:
   void f() {
      cout<<"Base\n";
      }
};
class Derived:public Base {
public:
   void f() {
      cout<<"Derived\n";
      }
};
main() {
   Derived obj;
   obj.Base::f();
}
```

```
The base output
```

51) What is the output of the following program?

```
#include<isotream>
using namespace std;
main(){
   char *s = "Fine";
     *s = 'N';

         cout<<s<<endl;
}

Runtime error
```

52) What is the output of the following program?

```
#include<isotream>

using namespace std;
class abc {
   public:
      static int x;
      int i;
      abc() {
         i = ++x;
         }
   };
int abc::x;

main() {
   abc m, n, p;

   cout<<m.x<<" "<<m.i<<endl;
}
```

A - 3 1

53) Which is the storage specifier used to modify the member variable even though the class object is a constant object?

mutable is storage specifier introduced in C++ which is not available in C. A class member declared with mutable is modifiable though the object is constant.

54) What is the output of the following program?

```
#include<isotream>
using namespace std;
main() {
  const int a = 5;

  a++;
  cout<<a;
}
```

55) What is the size of the following union definition?

```
#include<isotream>

using namespace std;
main()
{
  union abc {
    char a, b, c, d, e, f, g, h;

    int i;
    };
    cout<<sizeof(abc);
}
```

the size is 4

union size is biggest element size of it. All the elements of the union share common memory.

56) What is the output of the following program?

```
#include<isotream>

using namespace std;
int x = 5;

int& f() {
        return x;
}
main() {
        f() = 10;
   cout<<x;
}
```

A function can return reference, hence it can appear on the left hand side of the assignment operator.

57) What is the output of the following program?

```
#include<isotream>

using namespace std;
main()
{
  float t = 2;

        switch(t)
        {
                case 2: cout<<”Hi”;
                default: cout<<"Hello";
        }
}
```

D - Error

Error, switch expression can’t be float value

58) What is copy constructor?
A: Constructor which initializes the it's object member variables (by shallow copying) with another object of the same class. If you don't implement one in your class then compiler implements one for you. for example:
(a) Boo Obj1(10); // calling Boo constructor
(b) Boo Obj2(Obj1); // calling boo copy constructor
(c) Boo Obj2 = Obj1;// calling boo copy constructor

59) What is conversion constructor?
A: constructor with a single argument makes that constructor as conversion ctor and it can be used for type conversion.
for example:
class Boo
{
public:
Boo(int i);
};
Boo BooObject = 10 ; // assigning int 10 Boo object

60) Why an array always starts with index zero ?
The name of an array is essentially a pointer, a reference to a memory location, and so the expression array[n] refers to a memory location n-elements away from the starting element. This means that the index is used as an offset. The first element of the array is exactly contained in the memory location that array refers (0 elements away), so it should be denoted as array[0]

61) Where is memory for class-object allocated?
```
Class a; //stack. Usage: a.somethingInsideOfTheObject
Class *a = new Class(); //heap. Usage: a->somethingInsideOfTheObject
```
Note that if the class itself is allocating something on the heap, that part will always be on the heap, for example:
```
class MyClass{
public:
    MyClass()
    {
        a = new int();
    }
private:
    int * a;
};

void foo(){
    MyClass bar;
}
```
in this case the bar variable will be allocated on the stack, but the a inside of it will be allocated on the heap.
All local variables, no matter if from built-in types of from classes, or if they are arrays, are on the stack. All dynamic allocations are on the heap.

62) What is the maximum size that an array can hold?
The maximum size of an array is determined by the amount of memory that a program can access.

63) Stack vs HeapThe Stack
What is the stack? It's a special region of your computer's memory that stores temporary variables created by each function (including the `main()` function). The stack is a "FILO" (first in, last out) data structure, that is managed and optimized by the CPU quite closely. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, **all** of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.
The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use `malloc()` or `calloc()`, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using `free()` to deallocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a **memory leak**. That is, memory on the heap will still be set aside (and won't be available to other processes). As we will see in the debugging section, there is a tool called `valgrind` that can help you detect memory leaks.

64) What is the difference between the deep copy and shallow copy?
Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share the individual elements. Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated.
A *shallow copy* of an object copies all of the member field values. This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory. The pointer will be copied. but the memory it points to will not be copied -- the field in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The default copy constructor and assignment operator make shallow copies.
A *deep copy* copies all fields, *and* makes copies of dynamically allocated memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disasterous consequences.

65) What is the difference between structure and union?
With a union, you're only supposed to use one of the elements, because they're all stored at the same spot. This makes it useful when you want to store something that could be one of several types. A struct, on the other hand, has a separate memory location for each of its elements and they all can be used at once.

```
union foo {
  int a;   // can't use both a and b at once
  char b;
} foo;
struct bar {
 int a;   // can use both a and b simultaneously
 char b;
} bar;
union foo x;
x.a = 3; // OK
```

```
x.b = 'c'; // NO! this affects the value of x.a!

struct bar y;
y.a = 3; // OK
y.b = 'c'; // OK

union foo x;
x.a = 3;
x.b = ‘c';
printf("%i, %i\n", x.a, x.b);
prints:   99, 99  (c is 99)
```

66) ++i, i++ which one is faster ?

```
i++ is:
template<class T>
T foo (T& i) {
T temp(i);
i = i+1;
return temp;
}

++i is:
template<class T>
T& foo(T& i) {
i = i+1;
return i; }
```

++i is faster than i++. the latter involves a temporary object because
it must return the old value/object of i. for this reason, it’s generally better to use ++i.

67) What’s the difference between a linked list and an array?
A linked list can grow and shrink dynamically and you can add and delete nodes, we need to loop using iterate to access the elements in it.
A array size is fixed, you can access the element using the index. The memory location is one follow another.

68) What is the difference between a vector, a list and a map?
vector is a an array with dynamic size, you can access the element using the index.
You can grow and shrink dynamically for the list. Can not access the elements directly.
A map is a can be used to store element with key value.

69) Do you use any debug tools ?
Yes, I do have experience with Valgrind, and gdb for the debug.
For example, I always use valgrind to detect the memory leakage. It will generated a log file called valgrind.log, which contains the memory debug information at the bottom.
With gdb, you can print the variable on command line, you can setup one or more break point in the code, to do debug the code line by line.

valgrind -v --num-callers=40 --leak-check=full --error-limit=no --log-file=valgrind.log --suppressions=$ROOTSYS/root.supp --leak-resolution=high root.exe

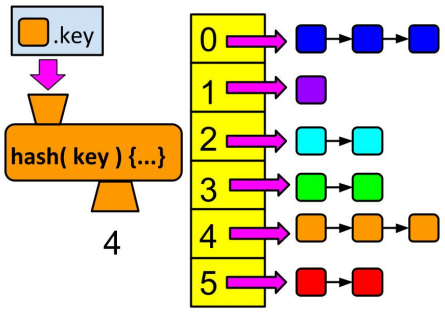
in valgrind.log:

==24895== Memcheck, a memory error detector
==24895== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==24895== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info

for gdb:
gdb test.exe
> break <line number>
> run
> list
> q

70) What is the difference between abstract class and interface ?
1 - interfaces can have no state or implementation
2 - a class that implements an interface must provide an implementation of all the method of that interface
3 - abstract classes may contain state (data members) and/or implementation (methods)
4 - abstract classes can be inherited without implementing the abstract methods (though such a derived class is abstract itself)
5 -interfaces may be multiple-inherited, abstract classes may not (**this is probably the key concrete reason for interfaces to exist separately from abtract classes** - they permit an implementation of multiple inheritance that removes many of the problems of general MI).

71) Hash table ?
Hash table is a structure which looking up element with index or key. You can get the element with a key and the hash function.




```
#include "HashTable.h"
// Constructs the empty Hash Table object.
// Array length is set to 13 by default.
HashTable::HashTable( int tableLength ) {
    if (tableLength <= 0) tableLength = 13;
    array = new LinkedList[ tableLength ];
    length = tableLength;
```

```
}

// Returns an array location for a given item key.
int HashTable::hash( string itemKey ) {
    int value = 0;
    for ( int i = 0; i < itemKey.length(); i++ )
        value += itemKey[i];
    return (value * itemKey.length() ) % length;
}

// Adds an item to the Hash Table.
void HashTable::insertItem( Item * newItem ) {
    int index = hash( newItem -> key );
    array[ index ].insertItem( newItem );
}

// Deletes an Item by key from the Hash Table.
// Returns true if the operation is successful.
bool HashTable::removeItem( string itemKey ) {
    int index = hash( itemKey );
    return array[ index ].removeItem( itemKey );
}

// Returns an item from the Hash Table by key.
// If the item isn't found, a null pointer is returned.
Item * HashTable::getItemByKey( string itemKey ) {
    int index = hash( itemKey );
    return array[ index ].getItem( itemKey );
}

// Display the contents of the Hash Table to console window.
void HashTable::printTable() {
    cout << "\n\nHash Table:\n";
    for ( int i = 0; i < length; i++ )
    {
        cout << "Bucket " << i + 1 << ": ";
        array[i].printList();
    }
}

// Prints a histogram illustrating the Item distribution.
void HashTable::printHistogram() {
    cout << "\n\nHash Table Contains ";
    cout << getNumberOfItems() << " Items total\n";
    for ( int i = 0; i < length; i++ )
    {
        cout << i + 1 << ":\t";
        for ( int j = 0; j < array[i].getLength(); j++ )
            cout << " X";
        cout << "\n";
    }
}

// Returns the number of locations in the Hash Table.
```

```
int HashTable::getLength() {
    return length;
}

// Returns the number of Items in the Hash Table.
int HashTable::getNumberOfItems() {
    int itemCount = 0;
    for ( int i = 0; i < length; i++ )
    {
        itemCount += array[i].getLength();
    }
    return itemCount;
}

// De-allocates all memory used for the Hash Table.
HashTable::~HashTable() {
    delete [] array;
}
```

72) list in std is a container that you can used to store object.

```
> list <int> a
>
#include <iostream>
#include <list>
int main (){
// constructors used in the same order as described above:
  std::list<int> first;                                // empty list of ints
  std::list<int> second (4,100);                       // four ints with value 100
  std::list<int> third (second.begin(),second.end());  // iterating through second
  std::list<int> fourth (third);                       // a copy of third
  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  std::list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are: “;
  for (std::list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
    std::cout << *it << ' ‘;
  std::cout << ‘\n';
  return 0;
}
```

73) An example Linkedlist ?

```
#include <iostream>
using namespace std;
class LinkedList{
    // Struct inside the class LinkedList
    // This is one node which is not needed by the caller. It is just
    // for internal work.
    struct Node {
        int x;
```

```
        Node *next;
    };
public:
   LinkedList(){
        head = NULL; // set head to NULL
    }
    // This prepends a new value at the beginning of the list
    void addValue(int val){
        Node *n = new Node();   // create new Node
        n->x = val;             // set value
        n->next = head;         // make the node point to the next node.
                                //  If the list is empty, this is NULL, so the end of the list --> OK
        head = n;               // last but not least, make the head point at the new node.
    }
    // returns the first element in the list and deletes the Node.
    // caution, no error-checking here!
    int popValue(){
        Node *n = head;
        int ret = n->x;
        head = head->next;
        delete n;
        return ret;
    }
private:
    Node *head; // this is the private member variable. It is just a pointer to the first Node
};

int main() {
    LinkedList list;
    list.addValue(5);
    list.addValue(10);
    list.addValue(20);
    cout << list.popValue() << endl;
    cout << list.popValue() << endl;
    cout << list.popValue() << endl;
    // because there is no error checking in popValue(), the following
    // is undefined behavior. Probably the program will crash, because
    // there are no more values in the list.
    // cout << list.popValue() << endl;
    return 0;
}
```

75) std::<array> is fixed-size sequence containers. You can access it with iterate or directly access it with the index.

```
array<int, 6> myArr = {4, 6, 6, 9, 10, 399}
for(auto it=myArr.begin(); it != myArr.end(); ++it) cout<<"  "<< *it;
for(int it=0; it<6; ++it) cout<<"  "<< myArr.at(it);
for(array<int, 6>::iterator iter=myArr.begin();  iter!= myArr.end(); ++iter) cout<<"  "<< *iter;
```

For normal array, int a[5]={}; When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty `[ ]`. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces `{ }`:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

76) Compare to std::<array>, std::<vector> is an array which can dynamically change in size. Both of them can use iterator and index to get the element.

```
vector<int> myVec;
for(int i=0; i<11; ++i) myVec.push_back(100-i);
vector<int>::iterator iterv;
for(iterv=myVec.begin(); iterv!=myVec.end(); ++iterv) cout<<"  "<<*iterv ;
```

77) std::<map> and <multmap>: Maps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order. In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a pair type combining both:

Associative: Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

Ordered: The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

Map: Each element associates a *key* to a *mapped value*: Keys are meant to identify the elements whose main content is the *mapped value*.

Unique keys: No two elements in the container can have equivalent *keys*.

Allocator-aware: The container uses an allocator object to dynamically handle its storage needs.

```
map<char,int> mymap;
   mymap['b'] = 100;
   mymap['a'] = 200;
   mymap['c'] = 300;
   cout<<"myMulSetStr using iterator:\n";
   for(map<char,int>::iterator iter=mymap.begin(); iter!=mymap.end(); ++iter) cout<<iter->first<<" => "<<iter->second <<endl;
```

The output is: a=>100 b=>200 c=>300

<multmap>:

```
std::multimap<char,int> mymultimap;
  mymultimap.insert (std::pair<char,int>('a',10));
  mymultimap.insert (std::pair<char,int>('b',20));
  mymultimap.insert (std::pair<char,int>('b',150));
  // show content:
  for (std::multimap<char,int>::iterator it=mymultimap.begin(); it!=mymultimap.end(); ++it)
std::cout << (*it).first << " => " << (*it).second << '\n';
```

The output is: a=>10 b=> 20 b=>150

78) std::<set> is containers that store unique elements following a specific order. In <set> an element is also a key to identify it.

Associative: Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

Ordered: The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.
Set: The value of an element is also the *key* used to identify it.
Unique keys: No two elements in the container can have equivalent *keys*.
Allocator-aware: The container uses an allocator object to dynamically handle its storage needs.

```
set<string> mySetStr;
    mySetStr.clear();
    const string name[5] = {"B", "A", "Zha", "Zhn", “Xue"};
    for(int i=0; i<5; i++) mySetStr.insert(name[i]);
    cout<<"mySet using iterator:”;
    for(set<string>::iterator iter=mySetStr.begin(); iter!=mySetStr.end(); ++iter) cout<<"
“<<*iter;
```

Ordered output: A, B, Xue, Zha, Zhn

79) std::<set> vs <multiset>
Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values.
associative: Elements in associative containers are referenced by their *key* and not by their absolute position in the container.
Ordered: The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.
Set: The value of an element is also the *key* used to identify it.
Multiple equivalent keys: Multiple elements in the container can have equivalent *keys*.
Allocator-aware: The container uses an allocator object to dynamically handle its storage needs.

```
multiset<string> myMulSetStr;
    myMulSetStr.clear();
    const string multname[5] = {"Liu", "Xue", "Liu", "Zhao", “Xue"};
    for(int i=0; i<5; i++) myMulSetStr.insert(multname[i]);
    cout<<"myMulSetStr using iterator:”;
    for(multiset<string>::iterator iter=myMulSetStr.begin(); iter!=myMulSetStr.end(); ++iter)
cout<<"  “<<*iter;
```

The output is: Liu Liu Xue Xue Zhao

```
auto itmstr = myMulSetStr.find(“Xue”);  /// remove first “Xue”
myMulSetStr.erase(itmstr);
for(multiset<string>::iterator iter=myMulSetStr.begin(); iter!=myMulSetStr.end(); ++iter)
cout<<"  “<<*iter;
```

The output is: Liu Liu Xue Zhao

80) std::<list>
Sequence: Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.
Doubly-linked list (bi-direction iterator): Each element keeps information on how to locate the next and the previous elements, allowing constant time insert and erase operations before or after a specific element (even of entire ranges), but no direct random access.

Allocator-aware: The container uses an allocator object to dynamically handle its storage needs.

```
list<string> myList;
    myList.push_back(“Liang");
    myList.push_back(“Xue");
    myList.push_back(“Li");
    myList.push_back(“Zhao");
    myList.push_back(“Liu");
    for(list<string>::iterator iter=myList.begin(); iter!=myList.end(); ++iter) cout<<" "<<*iter
<<endl;
```

The output is : Liang Xue Li Zhao Liu

81) std::<stack>
Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.
1 => top pop out
2
3

```
std::stack<int> mystack;
  for (int i=0; i<5; ++i) mystack.push(i);
  std::cout << "Popping out elements…";
  while (!mystack.empty()) {
    std::cout << ' ' << mystack.top();      ///// return a reference to the top element
    mystack.pop();    ////// remove the top element
  }
```

The output is: 4 3 2 1 0

82) std::<queue> **queue**s are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

```
 queue<int> myQueue;
    for(int i=1; i<10; i++) myQueue.push(i);
    cout<<"myQueue using while loop poping out element:”;
    while(!myQueue.empty()){
        cout<<"  “<<myQueue.front();
        myQueue.pop();
    }
```

The output is: 1 2 3 4 5 6 7 8 9 10

83) deque (双端队列) an irregular acronym of **d**ouble-**e**nded **que**ue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).
Sequence: Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.
Dynamic array: Generally implemented as a dynamic array, it allows direct access to any element in the sequence and provides relatively fast addition/removal of elements at the beginning or the end of the sequence.

Allocator-aware: The container uses an allocator object to dynamically handle its storage needs.

```
for (int i=1; i<=5; i++) mydeque.push_back(i);
  std::cout << "mydeque contains:";
  std::deque<int>::iterator it = mydeque.begin();
  while (it != mydeque.end())
    std::cout << ' ' << *it++;
```

84) Container summary:

1. std::<array> is fixed-size sequence containers, access with iterator, [i], at(i)
2. std::<vector> is an array which can dynamically change in size, access with iterator, [i], at(i)
3. std::<map> are associative, ordered containers that store elements formed by a combination of a *key value* and a *mapped value,* **No two elements in the container can have equivalent *keys***
4. *std::<multmap>, compare std::<map>, multmap can* ***have*** **equivalent *keys***
5. std::<set> are associative, ordered containers that store unique elements. In <set> an **element is also a key to identify it**. **Unique keys**: No two elements in the container can have equivalent *keys*.
6. std::<multiset>, compare to std::<set>, multiset can **have equivalent *keys*** *ie. elements*
7. std::<list> **Sequence:** Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence. **Doubly-linked list (bi-direction iterator):** Each element keeps information on how to locate the next and the previous elements, allowing constant time insert and erase operations before or after a specific element (even of entire ranges), but no direct random access. **Allocator-aware:** The container uses an allocator object to dynamically handle its storage needs.
8. std::<stack> Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.
9. std::<queue> **queue**s are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.
10. deque (双端队列) an irregular acronym of **d**ouble-**e**nded **que**ue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Makefile:
Makefile is a text file, that tells “make” how to compile and link a program. You can specify which libraries will be used for compilation in the makefile.
Target: target is a keyword to specify which part of command will be executed. For example, “all”, “clean”

Command line must start with tabs

Algorithms:

1) Suppose you had eight identical balls. One of them is slightly heavier and you are given a balance scale . What's the fewest number of times you have to use the scale to find the heavier ball?”
2) GIven 9 balls all of which weigh the same except for one, what is the minimum of weighings necessary to find the ball weighs more (or less).”
Divide the balls into three groups:
A: (Aa, Ab, Ac); B(Ba, Bb, Bc); C(Ca, Cb)

```
if(A>B){
   if(Aa>Ab) return Aa;
   else if (Aa<Ab) return Ab;
   else return Ac;
}
else if(A<B){
   if(Ba>Bb) return Ba;
   else if (Ba<Bb) return Bb;
   else return Bc
}
else {
    if(Ca>Cb) return Ca;
    else return Cb;
}
```

Punctuation

apostrophe ’ '
brackets [ ] ( ) { } 〈 〉
colon :
comma , ، 、
dash - – — —
ellipsis … ... . . .
exclamation mark !
full stop, period .
hyphen ‐
hyphen-minus -
question mark ?
quotation marks ‘ ’ “ ” ' ' " "
semicolon ;
slash, stroke, solidus / ⁄