# C++ Interview Questions and Answers

Compilation of many online sources

Gregg Roeten

Senior level interview

1. **Linked Lists: Why, When use**

   Remove duplicates from an unsorted linked lists.

   don't use any temporary buffers.

   Find the kth to last element of a linked list.

   Determine if linked list is circular.

   Reverse a linked list.

2. **When would you use a linked list vs. a vector?**

**linked list:**

**+ dynamic growth requirement,**

**+ remove element,**

- searches

**vector:**

**+ dynamic growth requirement,**

- remove element,

3. **Can you implement a Map with a tree?**

   Yes. Binary search tree, take O(log(n)) time to look up

4. **What about with a list?**

   Yes. Linked list, take O(n) time to look up. use a hash function to compute the key and each value is a list of tuple with(key, value). So it takes O(n) time to look up this list.

5. **How do you print out the nodes of a tree in level-order (i.e. first level, then 2nd level, then 3rd level, etc.)**

   Use BFS. record the "\n" index at each level

6. **dynamic arrays**
7. **hash tables/maps**
8. **Sorting: Compare and contrast different sorting algorithms. What are each's pros and cons? Which one is best for a given situation?**
9. **Project and Puzzle**

10. **Cognitive process**

    Hiring managers want to see the cognitive process that applicants go through when working on a problem or a puzzle.

    Always talk out loud when answering,

    address the pros and cons of your algorithm you've written,

    talk about space/time complexity in terms of Big-O.

    Breadth first search, depth first search,

    binary search, merge sort, quick sort, tree insert/find/etc

    Time/Space Complexity:

    Analyze time/space complexity in terms of Big-O. Answering this question shows the interviewer that you can intelligently talk about your solution, and consider other possible solutions as well.

11. **Arrays/Strings:**

    a. Determine if string has all unique characters. Next, don't use any additional data structures.

    b. Reverse a string.

    c. Given two strings, determine if one is a permutation of the other.

    d. Reverse words in a string.

12. **Stacks/Queue**

    Design a stack that keeps track of min/max element. O(1) time operations for maintaining the min/max.

    Solve Tower of Hanoi using Stacks.

    Implement a Queue using two Stacks.

13. **E. Trees/Graphs:**

    Find if there's a route between two nodes.

    Find if a binary tree is balanced.

    Perform a pre-order, in-order or post-order traversal of a Tree.

    Return the height of a tree.

14. **Additional Example Questions (analyze the runtime complexity of your solution for each)**

    Merge two sorted linked list of integers.

    Write a function that takes an integer N as an argument and prints all of the prime numbers between 1 and N (inclusive) to stdout.

    Design a dialer for a feature phone.

15. **Recursion**

## Advanced preparation:

Especially at smaller companies, and in interviews for specialized roles or research positions, some questions don't fall into any of the above categories and are often considerably more difficult (e.g. dynamic programming, distributed systems, etc). For specialized roles, I would stress doing personal projects, making open-source contributions, doing research, taking classes in the subject, etc. For general software developer roles in companies known for difficult questions, I would stress staying active in competitive programming and gaining experience in any domains important to the particular company.

## Techniques for technical questions:

The first hurdle is that you should NOT jump the gun and begin coding until: (1) you are confident that the route you are planning to take will solve all or most of the problem and not just the first few lines of it; and (2) you have resolved any obvious ambiguities needed to solve the problem. If you start coding before completing this first step, it will come off as a red flag to the interviewer.

If you know of an obvious brute force solution and only have a hazy idea about a possibly better solution, start with the brute force solution and explain to the interviewer that you're starting with a simple brute force approach and will see if you can improve it after you have something working. Something is better than nothing, and incrementally improving your code will suggest you have a clear, algorithmic thought process.

You must speak clearly and calmly and keep talking to let the interviewer know what you're thinking, where you're going, and what problems you're encountering.

Always listen to an interviewer's hints.

If you're unsure about a specific function or feature of the language you're using that isn't common knowledge, ask the interviewer if you can assume that a certain functionality exists and behaves in certain way. Often the interviewer will allow you to make that assumption (even if it doesn't exist) with little or no penalty.

If you get stuck, tell the interviewer you are stuck and explain why.

When you are done, DO NOT worry about taking too long to test your code for correctness unless the interviewer CLEARLY indicates your solution is correct. The interviewer will usually only give you one chance to test for correctness.

**Code a function to do X**. These questions are almost always solvable via a standard computer science algorithm in fewer than 20 lines of code. Preparing for questions like these is relatively easy, since you just must be familiar with core CS concepts, and able to implement them on the spot. The list of topics I'd suggest reviewing are: **binary search, tree traversal (pre/in/post), sorting algorithms (merge/quick/and some O(n^2) ones), recursion/iteration, graph search, dynamic programming, breadth first search, depth first search, stacks, queues, hashtables, heaps, priority queues, and linked lists (single/doubly/circular).**

**Puzzles**. I have no idea how to prepare for these. Personally, I think they have a low signal/noise ratio and don't ask them, but many companies do.

**Describe your experience/contribution to X**. The interviewer is wondering two things. First, do you really understand the project in depth. Second, were you a primary contributor. Preparing for these questions is as simple as going through your resume and making sure you can talk about anything mentioned for a few minutes.

**Design a system to do X.** The interviewer is wondering whether you can design larger systems. Pay attention to any potential problems with scaling, bottlenecks, and tradeoffs involved in your design. For instance, if asked to design a website architecture, saying "Apache and MySQL" would be epic fail- you should talk about LBs, caching, distributed databases (or other storage), potentially computationally expensive services that need to be run in a separate tier, and so on. I have no great advice for preparing for these questions, except maybe to get a whiteboard and figure out how you would design things you see on the internet: How would you design google search? How would you design an IM?

**Personality Questions.** Fit questions are pretty common, but I have no great advice for things like "what's your biggest weakness". My hunch is they're low signal to noise, and may merely signal how good a candidate is at bullshitting.

I have interviewed several candidates specifically focusing on their C++ knowledge, and if there was one question that worked well to put peoples' knowledge of C++ on a gradient, it was this one:

## Fix this memory leak as robustly as you can:

```
void doSomething()
{
Foo* pFoo = new Foo();
[do some stuff]
}
```

+1 for putting delete pFoo at the end

+2 for putting pFoo in a std::auto_ptr
+3 for knowing what RAII is - the concept, if not the acronym
+4 for mentioning exception-safety guarantees of the auto_ptr
+5 for putting pFoo in a boost:shared_ptr
+6 for knowing when a shared_ptr might not be freed.
+7 for talking about garbage collection techniques to fix circular references

This always worked to show how long someone had been working with C++. This is one datapoint you can use to tell where you are in the scale of C++ knowledge.

Edit: I would recommend someone for hire at level 3 or above.+1 because I learned a few things. Thanks! – John Oct

Just out of curiosity... why would you prefer boost::shared_ptr over std::auto_ptr without more information? I would be much happier with a candidate that responded with "it depends on what is in [do some stuff]" myself. – D.Shawley Oct 15 '09 at 3:03

Indeed. If the auto_ptr would do but you wanted to avoid its pitfalls, one would use boost::scoped_ptr (or std::tr1::unique_ptr). – UncleBens Oct 15 '09 at 9:12

**std::auto_ptr is not copyable** - if you try to pass it by value to another function, that function will take ownership of the pointee and, since arguments go out of scope at the end of the function call, free it then. Probably not what you had in mind. This is because auto_ptr only takes a pointer in new and guarantees deletion when out of scope. Boost's shared_ptr can be copied, as it maintains an internal reference count, so passing it by value into a function does "what you expect" by incrementing the reference count. Only when the count goes to 0 does it free the pointee. –

This is also why the **scoped ptr is great - it can't be copied, period**. While the **auto_ptr has "transfer of ownership"** copy semantics, scoped_ptr has "this code doesn't compile" copy semantics. Much harder to use unintuitively. Kudos to UncleBens for that. – Matt Oct 20 '09 at 6:52

16. **Even if they're interviewing for a C++ position not all questions may be specific to C++.**

For example, I've been hit with questions related to the following all in the same set of interviews for a single C++ position:

- Algorithmic complexity of well known sort and search algorithms
- Multithreaded programming
- Multiprocess programming
- Sockets programming
- Software development philosophy / approach
- Software test and validation philosophy / approach
- Debugging
- Benchmarking
- Dynamic and static analysis of code (e.g. run-time memory leak detection vs compile-time)

In my case, the phone interview was part of a screening process to determine if I could take an online C/C++ knowledge test (e.g. through BrainBench). The online test results then determined if I would be flown out for on-site interviews, which also included more "hands-on" software development tests.

YMMV. A lot depends on what you claim on your resume, as well.

Interviewers often try to help you by giving you hints so that they can see if you can arrive at the answer they're looking for. Besides gauging your knowledge, they also want to see how you think. Occassionally you might get a crummy interviewer that is neither helpful nor positive. The key is to be confident in your abilities and be truthful.

## Phone Interview questions:

Some coding question examples:

17. **Find the first non-repeating character in a string.**

18. **Determine if two strings are anagrams.**

19. **Find the most freq 65%uent number in an int array.**

20. **Find the nth number in the fibonacci sequence.**

21. **Determine if each character is unique in a string.**

22. **Determine if a string is a palindrome.**

23. **Determine if a string is sorted (return true if either in ascending or descending order and false otherwise).**

    Some non-coding question examples:

24. **Describe what happens in as much detail as you can provide when you insert a key with an associated value into a hash table.**

25. **Describe strategies for handling hash collisions. What are the trade-offs for each strategy?**

26. **How would design an object model that represents "x" where x could be a traffic light, a file system, etc.**

27. **Given simple function prototype with these parameters that returns this type of value, how would test an implementation of the prototype for correctness?**

28. **Write a short code using C++ to print out all odd number from 1 to 100 using a for loop(Asked by Intacct.com people)**

```
for( unsigned int i = 1; i < = 100; i++ )
  if( i & 0x00000001 )
    cout << i<<\",\";
```

    ISO layers and what layer is the IP operated from?( Asked by Cisco system people) Presentation, Session, Transport, Network, Data link and Physical. The IP is operated in the Network layer.

29. **Write a program that ask for user input from 5 to 9 then calculate the average( Asked by Cisco system people)**

```
A. int main()
   {
     int MAX=4;
     int total =0;
     int average=0;
     int numb;
     cout<<"Please enter your input from 5 to 9";
     cin>>numb;
     if((numb <5)&&(numb>9))
         cout<<"please re type your input";
     else
         for(i=0;i<=MAX; i++)
         {
            total = total + numb;
            average= total /MAX;
         }
     cout<<"The average number is"<<average<<endl;
     return 0;
     }
```

30. **Can you be bale to identify between Straight- through and Cross- over cable wiring? and in what case do you use Straight- through and Cross-over? (Asked by Cisco system people)**

    A. Straight-through is type of wiring that is one to to one connection Cross- over is type of wiring which those wires are got switched

We use Straight-through cable when we connect between NIC Adapter and Hub. Using Cross-over cable when connect between two NIC Adapters or sometime between two hubs.

31. **If you hear the CPU fan is running and the monitor power is still on, but you did not see any thing show up in the monitor screen. What would you do to find out what is going wrong? (Asked by WNI people)**

    A. I would use the ping command to check whether the machine is still alive(connect to the network) or it is dead.

32. **What is polymorphism?**

    Polymorphism is the idea that a base class can be inherited by several classes. A base class pointer can point to its child class and a base class array can store different child class objects.

33. **How to tell if you're on a UNIX system?**

    echo $SHELL

    OSx terminal, then the shell is specified in the Terminal's title bar

    You can do the Echo $RANDOM. It will return a undefined variable if you are from the C-Shell, just a return prompt if you are from the Bourne shell, and a 5 digit random numbers if you are from the Korn shell. You could also do a ps -l and look for the shell with the highest PID.

34. **What is Boyce Codd Normal form?**

    A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F+ of the form a->b, where a and b is a subset of R, at least one of the following holds:

    a->b is a trivial functional dependency (b is a subset of a)

    a is a superkey for schema R

35. **Tell how to check whether a linked list is circular.**

    As has been mentioned in other comments, the cycle doesn't necessarily contain the head/start node.

    A: tortoise and hare algorithm

    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\");
 }
}
```

36. **OK, why does this work?**

    If a list is circular, at some point pointer2 will wrap around and be either at the item just before pointer1, or the item before that. Either way, it's either 1 or 2 jumps until they meet.

37. **Explain which of the following declarations will compile and what will be constant - a pointer or the value pointed at:**

    const char *

    char const *

    char * const

    *Note:* Ask the candidate whether the first declaration is pointing to a string or a single character. Both explanations are correct, but if he says that it's a single character pointer, ask why a whole string is initialized as char* in C++. If he says this is a string declaration, ask him to declare a pointer to a single character. Competent candidates should not have problems pointing out why const char* can be both a character and a string declaration, incompetent ones will come up with invalid reasons.

38. **You’re given a simple code for the class BankCustomer. Write the following functions:**

Copy constructor

= operator overload

== operator overload

+ operator overload (customers’ balances should be added up, as an example of joint account between husband and wife)

*Note:*Anyone confusing assignment and equality operators should be dismissed from the interview. The applicant might make a mistake of passing by value, not by reference. The candidate might also want to return a pointer, not a new object, from the addition operator. Slightly hint that you’d like the value to be changed outside the function, too, in the first case. Ask him whether the statement customer3 = customer1 + customer2 would work in the second case.

39. **What problems might the following macro bring to the application?**

```
#define sq(x) x*x
```

40. **Consider the following struct declarations:**

```
struct A { A(){ cout << \"A\"; } };

struct B { B(){ cout << \"B\"; } };

struct C { C(){ cout << \"C\"; } };

struct D { D(){ cout << \"D\"; } };

struct E : D { E(){ cout << \"E\"; } };

struct F : A, B

{

    C c;

    D d;

    E e;

    F() : B(), A(),d(),c(),e() { cout << \"F\"; }

          };
```

What constructors will be called when an instance of F is initialized? Produce the program output when this happens.

Anything wrong with this code?

```
T *p = new T[10];

delete p;
```

*Note*: Incorrect replies: “No, everything is correct”, “Only the first element of the array will be deleted”, “The entire array will be deleted, but only the first element destructor will be called”.

41. **Anything wrong with this code?**

```
T *p = 0;

delete p;
```

*Note:* Typical wrong answer: Yes, the program will crash in an attempt to delete a null pointer. The candidate does not understand pointers. A **very smart** candidate will ask whether delete is overloaded for the class T.

42. **What’s potentially wrong with the following code?**

```
long value;

//some stuff

value &= 0xFFFF;
```

*Note:* Hint to the candidate about the base platform they’re developing for. If the person still doesn’t find anything wrong with the code, they are not experienced with C++.

43. **What does the following code do and why would anyone write something like that?**

```
void send (int *to, int * from, int count)
{
    int n = (count + 7) / 8;
    switch ( count  %  8)
    {
        case 0: do { *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
        } while ( --n > 0 );
    }
}
```

In the H file you see the following declaration:

```
class Foo {
void Bar( void ) const ;
};
```

44. **Tell me all you know about the Bar() function.**

45. **What is C++?**

Released in 1985, C++ is an object-oriented programming language created by Bjarne Stroustrup. C++ maintains almost all aspects of the C language, while simplifying memory management and adding several features - including a new datatype known as a class (you will learn more about these later) - to allow object-oriented programming. C++ maintains the features of C which allowed for low-level memory access but also gives the programmer new tools to simplify memory management.

46. **C++ used for:**

C++ is a powerful general-purpose programming language. It can be used to create small programs or large applications. It can be used to make CGI scripts or console-only DOS programs. C++ allows you to create programs to do almost anything you need to do. The creator of C++, Bjarne Stroustrup, has put together a partial list of applications written in C++.

47. **What is a class?**

Class is a user-defined data type.  A blueprint.  Created to solve a particular kind of problem. specifies the structure of data.

To use a Class you need to create objects out of the class.

- Class is static. All of the attributes of a class are fixed before, during, and after the execution of a program.

The attributes of a class don't change.

48. **What is an object**

- Object has a limited lifespan. Objects are created and eventually destroyed.
- during that lifetime, the attributes of the object may undergo significant change.

49. **What is the difference between an object and a class?**

Class attributes never change, whereas object attributes may undergo significant change.

50. **What is an access specifier?**

Access specifiers are used to define how the members (functions and variables) can be accessed outside the class. There are three access specifiers defined which are ***public, private, and protected***

**private:**
Members accessible only within the same class

**public:**
Members accessible from any where.

**protected:**
Members accessed from outside the class except a child class.

significance in inheritance.

51. **What is OOP's**

Polymorphism

Inheritance

Encapsulation

Abstraction

52. **What is Polymorphism**

call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

53. **What is Inheritance**

one class to inherit properties of another class

ie define a class in terms of another class,

allows code reuse

public

protected

private

54. **What is Encapsulation**

binds together the data and functions

keeps both safe from outside world.

Private Data members and public methods to access these data.

55. **What is Abstraction**

separation of interface and implementation.

hiding the internal implementations

56. **What is a modifier?**

A modifier, also called a modifying function is a member function that changes the value of at least one data member. In other words, an operation that modifies the state of an object. Modifiers are also known as 'mutators'. Example:

The function mod is a modifier in the following code snippet:

```
class test
{
    int x,y;
    public:
     test()
     {
        x=0; y=0;
     }
  void mod()
     {
         x=10;
         y=15;
     }
};
```

57. **What is an accessor?**
An accessor is a class operation that does not modify the state of an object. The accessor functions need to be declared as const operations

58. **Differentiate between a template class and class template.**
Template class: A generic definition or a parameterized class not instantiated until the client provides the needed information. It's jargon for plain templates.

Class template: A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed. It's jargon for plain classes.

59. **When does a name clash occur?**
A name clash occurs when a name is defined in more than one place. For example., two different class libraries could give two different classes the same name. If you try to use many class libraries at the same time, there is a fair chance that you will be unable to compile or link the program because of name clashes.

60. **Define namespace.**
It is a feature in C++ to minimize name collisions in the global name space. This namespace keyword assigns a distinct name to a library that allows other libraries to use the same identifier names without creating any name collisions. Furthermore, the compiler uses the namespace signature for differentiating the definitions

61. **What is the use of 'using' declaration.**
Introduces a name that is defined elsewhere into the declarative region where this using-declaration appears **.**

Using-declarations can be used to introduce namespace members into other namespaces and block scopes, or to introduce base class members into derived class definitions.

In namespace and block scope

Using-declaration introduces a member of another namespace into current namespace or block scope

```
#include <iostream>
#include <string>
using std::string;
int main()
{
    string str = "Example";
    using std::cout;
    cout << str;
}
```

A using declaration makes it possible to use a name from a namespace without the scope operator.

'using' declaration in C++ to add std::string and std::vector to the local namespace (to save typing unnecessary 'std::'s).

```
using std::string;
using std::vector;

class Foo { /*...*/ };
```

When you #include a header file in C++, it places the whole contents of the header file into the spot that you included it in the source file. So including a file that has a using declaration has the exact same effect of placing the using declaration at the top of each file that includes that header file.

The std namespace

The ANSI/ISO C++ standard requires you to explicitly declare the namespace in the standard library. For example, when using iostream, you must specify the namespace of **cout** in one of the following ways:

- std::cout (explicitly)
- using std::cout (using declaration)
- using namespace std (using directive)

62. **What is an Iterator class?**

A class that is used to traverse through the objects maintained by a container class. There are five categories of iterators:

- input iterators,
- output iterators,
- forward iterators,
- bidirectional iterators,
- random access.

An iterator is an entity that gives access to the contents of a container object without violating encapsulation constraints. Access to the contents is granted on a one-at-a-time basis in order. The order can be storage order (as in lists and queues) or some arbitrary order (as in array indices) or according to some ordering relation (as in an ordered binary tree). The iterator is a construct, which provides an interface that, when called, yields either the next element in the container, or some value denoting the fact that there are no more elements to examine. Iterators hide the details of access to and update of the elements of a container class.

The simplest and safest iterators are those that permit read-only access to the contents of a container class.

63. **What is an incomplete type?**

Incomplete types refers to pointers in which there is non availability of the implementation of the referenced location or it points to some location whose value is not available for modification.

```
int *i=0x400 // i points to address 400
*i=0; //set the value of memory location pointed by i.
```

Incomplete types are otherwise called uninitialized pointers.

11. What is a dangling pointer?

A dangling pointer arises when you **use the address of an object after its lifetime is over.** This may occur in situations like returning addresses of the automatic variables from a function or using the address of the memory block after

it is freed. The following code snippet shows this:

```
class Sample
{
public:
    int *ptr;
    Sample(int i)
    {
       ptr = new int(i);
    }
    ~Sample()
    {
        delete ptr;
    }
    void PrintVal()
    {
      cout << "The value is " << *ptr;
     }
 };
 void SomeFunc(Sample x)
 {
     cout << "Say i am in someFunc " << endl;
 }
 int main()
 {
    Sample s1 = 10;
    SomeFunc(s1);
    s1.PrintVal();
 }
```

In the above example when PrintVal() function is called it is called by the pointer that has been freed by the destructor in SomeFunc.

64. **Differentiate between the message and method.**

Message:

* Objects communicate by sending messages to each other.

* A message is sent to invoke a method.

Method

* Provides response to a message.

* It is an implementation of an operation

65. **What is an adaptor class or Wrapper class?**

A class that has no functionality of its own. Its member functions hide the use of a third party software component or an object with the non-compatible interface or a non-object-oriented implementation.

66. **What is a Null object?**

It is an object of some class whose purpose is to indicate that a real object of that class does not exist. One common use for a null object is a return value from a member function that is supposed to return an object with some specified properties but cannot find such an object.

67. **What is class invariant?**

A class invariant is a condition that defines all valid states for an object. It is a logical condition to ensure the correct working of a class. Class invariants must hold when an object is created, and they must be preserved under all operations of the class. In particular all class invariants are both preconditions and post-conditions for all operations or member functions of the class.

68. **What do you mean by Stack unwinding?**

It is a process during exception handling when the destructor is called for all local objects between the place where the exception was thrown and where it is caught.

69. **Define precondition and post-condition to a member function.**

Precondition: A precondition is a condition that must be true on entry to a member function. A class is used correctly if preconditions are never false. An operation is not responsible for doing anything sensible if its precondition fails to hold. For example, the interface invariants of stack class say nothing about pushing yet another element on a stack that is already full. We say that isful() is a precondition of the push operation.

Post-condition: A post-condition is a condition that must be true on exit from a member function if the precondition was valid on entry to that function. A class is implemented correctly if post-conditions are never false. For example, after pushing an element on the stack, we know that isempty() must necessarily hold. This is a post-condition of the push operation.

70. **What are the conditions that have to be met for a condition to be an invariant of the class?**

* The condition should hold at the end of every constructor.

* The condition should hold at the end of every mutator (non-const) operation.

71. **What are proxy objects?**

Objects that stand for other objects are called proxy objects or surrogates.

```
template <class t="">
class Array2D
{
public:
class Array1D
{
public:
T& operator[] (int index);
const T& operator[] (int index)const;
};

Array1D operator[] (int index);
const Array1D operator[] (int index) const;
};
```

The following then becomes legal:

```
Array2D<float>data(10,20);
cout<<data[3][6]; // fine
```

Here data[3] yields an Array1D object and the operator [] invocation on that object yields the float in position(3,6) of the original two dimensional array. Clients of the Array2D class need not be aware of the presence of the Array1D class. Objects of this latter class stand for one-dimensional array objects that, conceptually, do not exist for clients of Array2D. Such clients program as if they were using real, live, two-dimensional arrays. Each Array1D object stands for a one-dimensional array that is absent from a conceptual model used by the clients of Array2D. In the above example, Array1D is a proxy class. Its instances stand for one-dimensional arrays that, conceptually, do not exist.

72. **Name some pure object oriented languages.**
Smalltalk, Java, Eiffel, Sather.

73. **What is a node class?**
A node class is a class that,

* relies on the base class for services and implementation,

* provides a wider interface to the users than its base class,

* relies primarily on virtual functions in its public interface

* depends on all its direct and indirect base class

* can be understood only in the context of the base class

* can be used as base for further derivation

* can be used to create objects.

A node class is a class that has added new services or functionality beyond the services inherited from its base class.

74. **What is a container class? What are the types of container classes?**
A container class is a class that is used to hold objects in memory or external storage. A container class acts as a generic holder. A container class has a predefined behavior and a well-known interface. A container class is a supporting class whose purpose is to hide the topology used for maintaining the list of objects in memory. When a container class contains a group of mixed objects, the container is called a heterogeneous container; when the container is holding a group of objects that are all the same, the container is called a homogeneous container.

75. **What is polymorphism?**
Polymorphism is the idea that a base class can be inherited by several classes. A base class pointer can point to its child class and a base class array can store different child class objects.

76. **How can you tell what shell you are running on UNIX system?**
You can do the Echo $RANDOM. It will return a undefined variable if you are from the C-Shell, just a return prompt if you are from the Bourne shell, and a 5 digit random numbers if you are from the Korn shell. You could also do a ps -l and look for the shell with the highest PID.

77. **Write a Struct Time where integer m, h, s are its members**

```
struct Time
{
int m;
int h;
int s;
};
```

78. **How do you traverse a Btree in Backward in-order?**
Process the node in the right subtree

Process the root

Process the node in the left subtree

79. **In the derived class, which data member of the base class are visible?**
In the public and protected sections.

80. **What is the difference between realloc() and free()?**

The free subroutine frees a block of memory previously allocated by the new subroutine.

The realloc subroutine changes the size of the block of memory pointed to by the Pointer parameter to the number of bytes specified by the Size parameter and returns a new pointer to the block.

81. **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).

82. **What is the difference between declaration and definition?**

1) declaration

Introduces, Tells compiler later we'll define, describes its type, doesn't specify the functions body

2) Definition

contains the acutal implementation, Allot memory for it

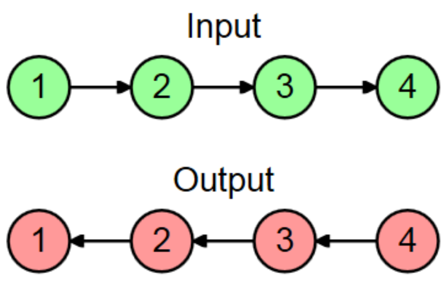
3) Initialization

assignment of a value at construction time

83. **How do you write a function that can reverse a linked-list?**

Is it a singly linked list of a doubly linked list?

**Singly linked.**



defining the data structure:

```
public class LinkedList {
    public int data;
    public LinkedList next;
}
```

The most important thing in this problem is to make sure the head and tail pointers are handled.

Notice in the image how the old head’s next point will be null whereas the new head is no longer null. The main section of the function will involve flipping the pointer between elements the other way.

## Pseudocode

```
function reverseSingly (head)
  if head is null
    return null
  prev ← null
  while head.next is not null
    head.next ← prev
    prev ← head
    head ← old value of head.next
  head.next ← prev
  return head
```

## C++ Code

```
public static LinkedList reverseSingly(LinkedList head) {
    if (head != null) {
        LinkedList prev = null;
        while (head.next != null) {
            LinkedList next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
        head.next = prev;
    }
    return head;
}
```

## 2nd solution     reverse a linked-list

```
void reverselist(void)
{
   if(head==0)
      return;
   if(head->next==0)
      return;
   if(head->next==tail)
   {
      head->next = 0;
      tail->next = head;
   }
   else
   {
      node* pre = head;
      node* cur = head->next;
      node* curnext = cur->next;
      head->next = 0;
      cur->next = head;
      for(; curnext!=0; )
      {
         cur->next = pre;
         pre = cur;
         cur = curnext;
         curnext = curnext->next;
```

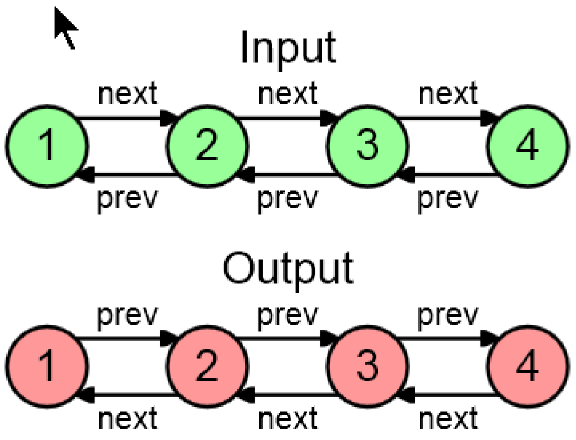
```
            }
            curnext->next = cur;
        }
    }
```

## Doubly Linked List

Analysis

Again, let's define the data structure:

```
public class LinkedList {
    public int data;
    public LinkedList next;
    public LinkedList prev;
}
```



It becomes clear from the picture that the list can be reversed by simply swapping each node's next and prev pointers. The only part that could be tricky with this question is ensuring that the new head pointer is valid and doesn't introduce a cycle.

### Pseudocode

```
function reverseDoubly (head)
  if head is null
    return null

  loop {
    swap head.next and head.prev values
    if head.prev is null
      return head
    head ← head.prev
  }
```

### Code

```
public static LinkedList reverseDoubly(LinkedList head) {
    while (head != null) {
        LinkedList temp = head.next;
        head.next = head.prev;
        head.prev = temp;
        if (temp == null)
            break;
        head = temp;
    }
    return head;
}
```

84. **How do you find out if a linked-list has an end? (i.e. the list is not a cycle, circular)**

You can find out by using 2 pointers.

One of them goes 2 nodes each time.

The second one goes at 1 nodes each time.

If there is a cycle, the one that goes 2 nodes each time will eventually meet the one that goes slower. If that is the case, then you will know the linked-list is a cycle.

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\");
   }
}
```

OK, why does this work?

If a list is circular, at some point pointer2 will wrap around and be either at the item just before pointer1, or the item before that. Either way, it’s either 1 or 2 jumps until they meet.

85. **Write a program that ask for user input from 5 to 9 then calculate the average**

```
#include "iostream.h"
int main() {
   int MAX = 4;
   int total = 0;
   int average;
   int numb;

   for (int i=0; i<MAX; i++) {
      cout << "Please enter your input between 5 and 9: ";
      cin >> numb;
      while ( numb<5 || numb>9) {
         cout << "Invalid input, please re-enter: ";
         cin >> numb;
      }
      total = total + numb;
   }
   average = total/MAX;
   cout << "The average number is: " << average << "\n";
   return 0;
}
```

86. **Write a short code using C++ to print out all odd number from 1 to 100 using a for loop**

```
for( unsigned int i = 1; i < = 100; i++ )
    if( i & 0x00000001 )
        cout << i << \",\";
```

87. **Write a function that swaps the values of two integers, using int* as the argument type.**

```
void swap(int* a, int*b) {
   int t;
   t = *a;
   *a = *b;
   *b = t;
}
```

88. **What is virtual constructors/destructors?**

Virtual destructors: answer #1

If an object (with a non-virtual destructor) is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the base-class destructor function (matching the pointer type) is called on the object.

There is a simple solution to this problem declare a virtual base-class destructor.

This makes all derived-class destructors virtual even though they don't have the same name as the base-class destructor. Now, if the object in the hierarchy is destroyed explicitly by applying the delete operator to a base-class pointer to a derived-class object, the destructor for the appropriate class is called. Virtual constructor: Constructors cannot be virtual. Declaring a constructor as a virtual function is a syntax error.

Virtual destructors: answer # 2

If an object (with a non-virtual destructor) is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the base-class destructor function (matching the pointer type) is called on the object.

There is a simple solution to this problem – declare a virtual base-class destructor. This makes all derived-class destructors virtual even though they don't have the same name as the base-class destructor. Now, if the object in the hierarchy is destroyed explicitly by applying the delete operator to a base-class pointer to a derived-class object, the destructor for the appropriate class is called.

Virtual constructor:

Constructors cannot be virtual. Declaring a constructor as a virtual function is a syntax error.

89. **Does c++ support multilevel and multiple inheritance?**

Yes.

90. **What are the advantages of inheritance?**

• It permits code reusability.

• Reusability saves time in program development.

• It encourages the reuse of proven and debugged high-quality software, thus reducing problem after a system becomes functional.

91. **What is the difference between an ARRAY and a LIST?**

Answer1

Array is collection of homogeneous elements.

List is collection of heterogeneous elements.

For Array memory allocated is static and continuous.

For List memory allocated is dynamic and Random.

Array: User need not have to keep in track of next memory allocation.

List: User has to keep in Track of next location where memory is allocated.

Answer2

Array uses direct access of stored members, list uses sequencial access for members.

//With Array you have direct access to memory position 5

```
Object x = a[5]; // x takes directly a reference to 5th element of
array
```

//With the list you have to cross all previous nodes in order to get the 5th node:

```
list mylist;
list::iterator it;
for( it = list.begin() ; it != list.end() ; it++ )
{
if( i==5)
{
x = *it;
break;
}
i++;
}
```

92. **What is a template?**

Templates allow to create generic functions that admit any data type as parameters and return value without having to overload the function with all the possible data types. Until certain point they fulfill the functionality of a macro. Its prototype is any of the two following ones:

template <class indetifier> function_declaration;

template <typename indetifier> function_declaration;

The only difference between both prototypes is the use of keyword class or typename, its use is indistinct since both expressions have exactly the same meaning and behave exactly the same way.

93. **Define a constructor - What it is and how it might be called (2 methods).**

constructor is a member function of the class,

with the name of the function being the same as the class name.

also specifies how the object should be initialized.

Ways of calling constructor:

1) **Implicitly**: automatically by complier when an object is created, allocate mem on STACK

2) **explicitly**: call constructor, allocats memory on HEAP

ex

```
class Point2D{
   int x; int y;
   public Point2D() : x(0) , y(0) {} //default  constructor
};

main(){
   Point2D MyPoint; // Implicit Constructor call memory on stack

   Point2D * pPoint = new Point2D(); // Explicit call,  on HEAP
```

94. **You have two pairs: new() and delete() and another pair : alloc() and free(). Explain differences between eg. new() and malloc()**

Answer1

| | |
|---|---|
| "new and delete" | malloc() and free()" |
| preprocessors | "are functions. |
| auto alloc memory | "malloc()" we have to use "sizeof()". |
| initlize the new memory to 0 | gives random value in the new alloted memory |

better to use calloc()

95. **What is the difference between class and structure?**

| Structure | Class |
|---|---|
| declarations inside a structure are by **default public**. | all the members inside the class are **default private**. |

Class is a successor of Structure.

96. **What is Runtime type identification RTTI?**
lets you find the dynamic type of an object when you have only a pointer or a reference to the base type.

RTTI is the official way in standard C++ to discover the type of an object and to convert the type of a pointer or reference (that is, dynamic typing).

.

97. **What is a COPY CONSTRUCTOR and when is it called?**
The copy constructor is used to:

1. **Initialize one object from another of the same type.**
2. **Copy an object to pass it as an argument to a function.**
3. **Copy an object to return it from a function.**

A copy constructor is a method that accepts an object of the same class and copies it's data members to the object on the left part of assignement:

```
class Point2D{
int x; int y;
public int color;
protected bool pinned;
public Point2D() : x(0) , y(0) {} //default (no argument) constructor
public Point2D( const Point2D & ) ;
};

Point2D::Point2D( const Point2D & p )
{
this->x = p.x;
this->y = p.y;
this->color = p.color;
this->pinned = p.pinned;
}
main(){
```

```
Point2D MyPoint;
MyPoint.color = 345;
Point2D AnotherPoint = Point2D( MyPoint );
// now AnotherPoint has color = 345
```

98. **What is the word you will use when defining a function in base class to allow this function to be a polimorphic function?**

    virtual

99. **What do you mean by binding of data and functions?**

    Encapsulation.

100. **How can you export a function from a DLL?**

    Use the keyword __declspec(dllexport) in the function's definition.

    **DLL** file has a layout very similar to an .exe file, with one important difference — a DLL file contains an exports table. The exports table contains the name of every function that the DLL exports to other executables. These functions are the entry points into the DLL; only the functions in the exports table can be accessed by other executables. Any other functions in the DLL are private to the DLL. The exports table of a DLL can be viewed by using the DUMPBIN tool with the /EXPORTS option.

101. **Suppose that data is an array of 1000 integers. Write a single function call that will sort the 100 elements data [222] through data [321].**

    quicksort ((data + 222), 100);

102. **Which recursive sorting technique always makes recursive calls to sort subarrays that are about half size of the original array?**

    Mergesort always makes recursive calls to sort subarrays that are about half size of the original array, resulting in O(n log n) time.

103. **What is abstraction?**

    Abstraction is of the process of hiding unwanted details from the user.

104. **What is the difference between an external iterator and an internal iterator? Describe an advantage of an external iterator.**

    An internal iterator is implemented with member functions of the class that has items to step through.

    An external iterator is implemented as a separate class that can be "attach" to the object that has items to step through.

105. **What are virtual functions – Write an example?**

    used with inheritance

    Virtual functions are member functions whose behavior can be overridden in derived classes.

    resolved late, at runtime. Virtual keyword is used to make a function virtual.

106. **Explain virtual inheritance. Draw the diagram explaining the initialization of the base class when virtual inheritance is used.**

    *Note:* Typical mistake for applicant is to draw an inheritance diagram, where a single base class is inherited with virtual methods. Explain to the candidate that this is not virtual inheritance. Ask them for the classic definition of virtual inheritance. Such question might be too complex for a beginning or even intermediate developer, but any applicant with advanced C++ experience should be somewhat familiar with the concept, even though he'll probably say he'd avoid using it in a real project.

Moreover, even the experienced developers, who know about virtual inheritance, cannot coherently explain the initialization process. If you find a candidate that knows both the concept and the initialization process well, he’s hired.

107. **What is pure virtual function?**

```
    class Shape { public: virtual void draw() = 0; };
```

108. **Early binding and late binding? How it is related to dynamic binding?**

Binding is the process of linking actual address of functions or identifiers to their reference. This happens mainly two times.

**During compilation : This is called early binding** all the direct function references compiler will replace the reference with actual address of the method.

**At runtime : This is called late binding.** In case of **virtual function** calls using a Base reference, compiler does not know which method will get called at run time. In this case compiler will replace the reference with code to get the address of function at runtime.

**Dynamic binding** is another name for late binding.

109. **When to use Pure Virtual Functions?**

when definition of a virtual function in the base class does not exist.

110. **Virtual Destructor - What is the need for â€œVirtual Destructorâ€?**

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 classâ€™s 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.

111. **What is polymorphism? Explain with an example?**

"Poly" means "many" and "morph" means "form". Polymorphism is the ability of an object (or reference) to assume (be replaced by) or become many different forms of object.

Example: function overloading, function overriding, virtual functions. Another example can be a plus ‘+’ sign, used for adding two integers or for using it to concatenate two strings.

112. **How can you quickly find the number of elements stored in a**

a) static array

C-style array, then you can do something like:

```
int a[7];
std::cout << "Length of array = " << (sizeof(a)/sizeof(*a)) <<
std::endl;
```

In C++, if you want this kind of behaviour, then you should be using a container class; probably std::vector or even std::array

b) dynamic array ?

Uses pointers therefore sizeof doesn’t work

it will be decided at runtime only

113. **Why is it difficult to store linked list in an array?**

linked list dynamic, array static

114. **Design a parking lot management system that can be used to find vacant slots"**

```
Ref C++ documents
```

115. **How can you find the nodes with repetetive data in a linked list?**

Write a prog to accept a given string in any order and flash error if any of the character is different. For example : If abc is the input then abc, bca, cba, cab bac are acceptable but aac or bcd are unacceptable.

Write out a function that prints out all the permutations of a string. For example, abc would give you abc, acb, bac, bca, cab, cba. You can assume that all the characters will be unique.

116. **What's the output of the following program? Why?**

```
#include <stdio.h>
main()
{
    typedef union
    {
        int a;
        char b[10];
        float c;
    }
    Union;

    Union x,y = {100};
    x.a = 50;
    strcpy(x.b,\"hello\");
    x.c = 21.50;

printf(\"Union x : %d %s %f \n\",x.a,x.b,x.c );
printf(\"Union y :%d %s%f \n\",y.a,y.b,y.c);
}
```

117. **Given inputs X, Y, Z and operations | and & (meaning bitwise OR and AND, respectively) What is output equal to in output = (X & Y) | (X & Z) | (Y & Z)**

118. **What is an HTML tag?**

Answer: An HTML tag is a syntactical construct in the HTML language that abbreviates specific instructions to be executed when the HTML script is loaded into a Web browser. It is like a method in Java, a function in C++, a procedure in Pascal, or a subroutine in FORTRAN.

119. **Explain which of the following declarations will compile and what will be constant - a pointer or the value pointed at: * const char ***

* char const *

* char * const

Note: Ask the candidate whether the first declaration is pointing to a string or a single character. Both explanations are correct, but if he says that it's a single character pointer, ask why a whole string is initialized as char* in C++. If he says this is a string declaration, ask him to declare a pointer to a single character. Competent candidates should not have problems pointing out why const char* can be both a character and a string declaration, incompetent ones will come up with invalid reasons.

120. **You're given a simple code for the class BankCustomer. Write the following functions:**

* Copy constructor

* = operator overload

* == operator overload

* + operator overload (customers' balances should be added up, as an example of joint account between husband and wife)

Note:Anyone confusing assignment and equality operators should be dismissed from the interview. The applicant might make a mistake of passing by value, not by reference. The candidate might also want to return a pointer, not a new object, from the addition operator. Slightly hint that you'd like the value to be changed outside the function, too, in the first case. Ask him whether the statement customer3 = customer1 + customer2 would work in the second case.

121.**Anything wrong with this code?**

T *p = new T[10];

delete p;

Everything is correct, Only the first element of the array will be deleted", The entire array will be deleted, but only the first element destructor will be called.

122.**Anything wrong with this code?**

T *p = 0;

delete p;

Yes, the program will crash in an attempt to delete a null pointer.

123.**What's the best way to declare and define global variables?**

The best way to declare global variables is to declare them after including all the files so that it can be used in all the functions.

124.**What does extern mean in a function declaration?**

Using extern in a function declaration we can make a function such that it can used outside the file in which it is defined.

An extern variable, function definition, or declaration also makes the described variable or function usable by the succeeding part of the current source file. This declaration does not replace the definition. The declaration is used to describe the variable that is externally defined.

If a declaration for an identifier already exists at file scope, any extern declaration of the same identifier found within a block refers to that same object. If no other declaration for the identifier exists at file scope, the identifier has external linkage.

125.**What can I safely assume about the initial values of variables which are not explicitly initialized?**

It depends on complier which may assign any garbage value to a variable if it is not initialized.

126.**What is the difference between char a[] = "string"; and char *p = "string";?**

In the first case 6 bytes are allocated to the variable a which is fixed, where as in the second case if *p is assigned to some other value the allocate memory can change.

127.**What is the difference between char a[] = "string"; and char *p = "string"; ?**

Answer1

a[] = "string";

char *p = "string";

The difference is this:

p is pointing to a constant string, you can never safely say

p[3]='x';

however you can always say a[3]='x';

char a[]="string"; - character array initialization.

char *p="string" ; - non-const pointer to a const-string.( this is permitted only in the case of char pointer in C++ to preserve backward compatibility with C.)

Answer2

a[] = "string";

char *p = “string”;

a[] will have 7 bytes. However, p is only 4 bytes. P is pointing to an adress is either BSS or the data section (depending on which compiler — GNU for the former and CC for the latter).

Answer3

char a[] = “string”;

char *p = “string”;

for char a[]…….using the array notation 7 bytes of storage in the static memory block are taken up, one for each character and one for the terminating nul character.

But, in the pointer notation char *p…………the same 7 bytes required, plus N bytes to store the pointer variable “p” (where N depends on the system but is usually a minimum of 2 bytes and can be 4 or more)……

128. **How do I declare an array of N pointers to functions returning pointers to functions returning pointers to characters?**

Answer1

If you want the code to be even slightly readable, you will use typedefs.

typedef char* (*functiontype_one)(void);

typedef functiontype_one (*functiontype_two)(void);

functiontype_two myarray[N]; //assuming N is a const integral

Answer2

char* (* (*a[N])())()

Here a is that array. And according to question no function will not take any parameter value.

129. **How do I initialize a pointer to a function?**

This is the way to initialize  a pointer to a function

```
void fun(int a)
{
}
void main()
{
void (*fp)(int);
fp=fun;
fp(1);
}
```

This is the way to initialize a pointer to a function

void fun(int a) { } void main() { void (*fp)(int); fp=fun; fp(1); }

130. **How does throwing and catching exceptions differ from using setjmp and longjmp?**

The throw operation calls the destructors for automatic objects instantiated since entry to the try block.

131. **What is a default constructor?**

Default constructor WITH arguments

class B {

```
public:
B (int m = 0) : n (m) {}
int n;
};
int main(int argc, char *argv[])
{
B b;
return 0;
}
```

132. **What is a conversion constructor?**

A constructor that accepts one argument of a different type.

133. **What is the difference between a copy constructor and an overloaded assignment operator?**

A copy constructor constructs a new object by using the content of the argument object. An overloaded assignment operator assigns the contents of an existing object to another existing object of the same class.

134. **When should you use multiple inheritance?**

There are three acceptable answers: "Never," "Rarely," and "When the problem domain cannot be accurately modeled any other way."

When you are designing a generic class to contain or otherwise manage objects of other types, when the format and behavior of those other types are unimportant to their containment or management, and particularly when those other types are unknown (thus, the genericity) to the designer of the container or manager class.

135. **What is a mutable member?**

One that can be modified by the class even when the object of the class or the member function doing the modification is const.

136. **What is an explicit constructor?**

A conversion constructor declared with the explicit keyword. The compiler does not use an explicit constructor to implement an implied conversion of types. It's purpose is reserved explicitly for construction.

137. **What is the Standard Template Library (STL)?**

A library of container templates approved by the ANSI committee for inclusion in the standard C++ specification.

A programmer who then launches into a discussion of the generic programming model, iterators, allocators, algorithms, and such, has a higher than average understanding of the new technology that STL brings to C++ programming.

138. **Describe run-time type identification.**

The ability to determine at run time the type of an object by using the typeid operator or the dynamic_cast operator.

139. **What is "mutable"?**

Answer1.

"mutable" is a C++ keyword. When we declare const, none of its data members can change. When we want one of its members to change, we declare it as mutable.

Answer2.

A "mutable" keyword is useful when we want to force a "logical const" data member to have its value modified. A logical const can happen when we declare a data member as non-const, but we have a const member function attempting to modify that data member. For example:

```
class Dummy {
  public:
    bool isValid() const;
  private:
    mutable int size_ = 0;
    mutable bool validStatus_ = FALSE;
      // logical const issue resolved
};
bool Dummy::isValid() const
  // data members become bitwise const
{
  if (size > 10) {
    validStatus_ = TRUE; // fine to assign
    size = 0; // fine to assign
  }
}
```

140. **STL Containers - What are the types of STL containers?**

There are 3 types of STL containers:

1. Adaptive containers like queue, stack
2. Associative containers like set, map
3. Sequence containers like vector, deque

141. **What is output equal to in Bitwise Operations - ...**

What is output equal to in Bitwise Operations - Given inputs X, Y, Z and operations | and & (meaning bitwise OR and AND, respectively), what is output equal to in?

output = (X & Y) | (X & Z) | (Y & Z);

142. **What's the meaning of ARP in TCP/IP?**

The "ARP" stands for Address Resolution Protocol. The ARP standard defines two basic message types: a request and a response. a request message contains an IP address and requests the corresponding hardware address; a replay contains both the IP address, sent in the request, and the hardware address.

143. **What is a Makefile?**

Makefile is a utility in Unix to help compile large programs. It helps by only compiling the portion of the program that has been changed.

A Makefile is the file and make uses to determine what rules to apply. make is useful for far more than compiling programs.

144. **What is Mutex (mutual exclusion semaphore)**

- **locking mechanism** for code that you only want to be executed one thread at a time, mutex owned by that process
- **Ex** Delete a node from a linked list.
  - Don't want another thread to change current pointers while deleting the node.
  - So you set mutex while you are deleting the node.
  - If another thread tries to acquire the same mutex, it will be put to sleep unitl youi release the

mutex.

145. **What is semaphore?**

- Allows one or more threads to share resource.
- Ex 3 tape drives
    - Semaphore is set to 3.
    - The count is decremented as threads use the tape drive. Restricting the number of simutaneous users of a shared resource.
- Semaphores are use for thread synchronization.
- binary semaphores (which are really semaphores with count 1).

146. **Can a thread acquire more than one lock (Mutex)?**

Yes, it is possible that a thread is in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

147. **Can a mutex be locked more than once?**

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a *recursive mutex* can be locked more than once (POSIX complaint systems),

148. **What happens if a non-recursive mutex is locked more than once.**

Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of mutex and return if it is already locked by same thread to prevent deadlocks.

149. **What we mean by "thread blocking on mutex/semaphore" when they are not available?**

Every synchronization primitive has a waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of processor to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list gets the resource (more precisely, it depends on the scheduling policies).

150. **What is the difference between Stack and Queue?**

Stack is a Last In First Out (LIFO) data structure.

151. **Write a fucntion that will reverse a string.**

```
char *strrev(char *s)
{
   int i = 1,
   len = strlen(s);

   char *str;
   if ((str = (char *)malloc(len+1)) == NULL) {
      /*cannot allocate memory */
      err_num = 2;
      return (str);
   else
   {
      while(len >= i) {
         str[i]=s[len];
         len--;
         i++;
      }
   return (str);
}
}
```

152. **What is the software Life-Cycle?**

The software Life-Cycle are

1) Analysis and specification of the task

2) Design of the algorithms and data structures

3) Implementation (coding)

4) Testing

5) Maintenance and evolution of the system

6) Obsolescence

153.**Describe Stacks and name a couple of places where stacks are useful.**

- LIFO insertions and deletions are always made at the top.
- A Stack is a linear structure
- USE when we need to check some syntex errors, such as missing parentheses.
- USE to track forms as a user opens them and then be able to back out of the open forms in the opposite order:

154.**What Is a Socket?**

- **IP address + a port number**
- represent the two-way link connection between a client program and a server program.
- The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively.

155.**Question 1: How does the race condition occur?**

It occurs when two or more processes are reading or writing some shared data and the final result depends on who runs precisely when.

156.**Question 2: What is multiprogramming?**

Multiprogramming is a rapid switching of the CPU back and forth between processes.

157.**QUESTION: What are the seven layers(OSI model) of networking?**

**7.Application**

**6.Presentation**

**5.Session**

**4.Transport**

**3.Network**

**2.Data Link**

**1.Physical**

158.**QUESTION: What is the difference between a NULL pointer and a void pointer? (asked by Lifescan inc)**

- NULL pointer is a pointer of any type whose value is zero.
- A void pointer is a pointer to an object of an unknown type, and is guaranteed to have enough bits to hold a pointer to any object. A void pointer is not guaranteed to have enough bits to point to a function (though in general practice it does).

159.**What is .NET?**

.NET is essentially a framework for software development.It is similar in nature to any other software development framework (J2EE etc) in that it provides a set of runtime containers/capabilities, and a rich set of pre-built functionality in the form of class libraries and APIs

The .NET Framework is an environment for building, deploying, and running Web Services and other applications. It consists of three main parts: the Common Language Runtime, the Framework classes, and ASP.NET.

160.**26. What is .NET Framework?**

It is a Framework in which Windows applications may be developed and run. The Microsoft .NET Framework is a platform for building, deploying, and running Web Services and applications. It provides a highly productive, standards-based, multi-language environment for integrating existing investments with next-generation applications and services as well as the agility to solve the challenges of deployment and operation of Internet-scale applications. The .NET Framework consists of three main parts: the common language runtime, a hierarchical set of unified class libraries, and a componentized version of Active Server Pages called ASP.NET. The .NET Framework provides a new programming model and rich set of classes designed to simplify application development for Windows, the Web, and mobile devices. It provides full support for XML Web services, contains robust security features, and delivers new levels of programming power. The .NET Framework is used by all Microsoft languages including Visual C#, Visual J#, and Visual C++.

161. **35. What is the difference between ASP and ASP.NET?**

ASP is interpreted.

ASP .NET

- code is not fully backward compatible with ASP.
- Compiled event base programming.
- better language support, a large set of new controls and XML based components, and better user authentication.
- increased performance by running compiled code.

162. **Which of the following library function below by default aborts the program?**

Terminate()Terminate is wrong, case sensitive, should be **terminate**

end()

Abort()

exit() and can be called to end the program

163. **What happens when a pointer is deleted twice**

abort the prog

cause a failure

cause error GR

cause trap If you delete a pointer and set it to NULL, it it possibly cannot have an adverse effect. It is safe. However, if a pointer is deleted an not nullified, then it can cause a trap.

164. **What is a deep copy**

creates a copy of the **dynamically alloted objects too**.

165. **What is a shallow copy**

creates a copy of the “copy constructor” provided by the compiler, **copies address of memory locatioin** instead of copying the content.

166. **By default all the struct members are public while by default class members are private.**

167. **Vtable**

creates a static table per class

creates a static table per object

creates a dynamic table per class

creates a dyunamic table per object

168. **Specify that the compiler should match function calls with the correct definition at run time.**

Static binding

Dynamic binding

169. **Iterator types**

Input Iterator

Backward Iterator

Forward Iterator

Both a and c

170. **Templates**

STL Standard Template

Container, Iterator, Algorithm

**Input Iterator:** These iterators can read values in the forward movement. They can be incremented, compared and dereferenced.

**Ouptut Iterator:** They write values in the forward movement. They can be incremented and dereferenced.

**Forward Iterator:** They are like input and output iterators that can read and write values in forward movement.

**Bidirectional iterators:** These can Read and write values with forward and backward movement and can be incremented, decremented.

**random_access_iterator:** Can read and write values randomly.

```
#include<iostream>
using namespace std;
class A
{
   int i;
public:
```

171. **// class A has a constructor which can be called with single integer argument, so an int can be converted to A**

```
   A(int ii = 0) : i(ii) {}
   void show() {  cout << i << endl;  }
};

class B
{
   int x;
public:
```

172. **// class B has as conversion operator overloaded, so an object of B can be converted to that of**

**A.**

```
    B(int xx) : x(xx) {}
    operator A() const {  return A(x); }
};

void g(A a)
{
    a.show();
}

int main()
{
    B b(10);
    g(b);
    g(20);
    return 0;
}
```

**(A)** Compiler Error
**(B)** 10 20
**(C)** 20 20
**(D)** 10 10

Answer: (B)

**Explanation:** Note that the class B has as conversion operator overloaded, so an object of B can be converted to that of A.

Also, class A has a constructor which can be called with single integer argument, so an int can be converted to A.

Question and Answer

```
void cp(char *a)
{
char *f,*p;
f=(char *)malloc(strlen(a)*sizeof(char));
p = f;
while(*a != '\0')              //previous code*a != 0
{
```

```
*f=*a;
f++;
a++;
}
*f = '\0';
printf("%s\n",p);
return f;             // f is local, can't return local var
}
```

## Design pattern interview questions for Senior and experienced level

173. **Give an example where you prefer abstract class over interface ?**
This is common but yet *tricky design interview question*. both interface and abstract class follow "writing code for interface than implementation" design principle which adds flexibility in code, quite important to tackle with changing requirement. here are some pointers which help you to answer this question:

1. In Java you can only extend one class but implement multiple interface. So if you extend a class you lost your chance of extending another class.

2. Interface are used to represent adjective or behavior e.g. Runnable, Clonable, Serializable etc, so if you use an abstract class to represent behavior your class can not be Runnable and Clonable at same time because you can not extend two class in Java but if you use interface your class can have multiple behavior at same time.

3. On time critical application prefer abstract class is slightly faster than interface.

4. If there is a genuine common behavior across the inheritance hierarchy which can be coded better at one place than abstract class is preferred choice. Some time interface and abstract class can work together also where defining function in interface and default functionality on abstract class.

174. **Design a Vending Machine which can accept different coins, deliver different products?**
This is an open design question which you can use as exercise, try producing **design document**, **code** and **Junit test** rather just solving the problem and check how much time it take you to come to solution and produce require artifacts, Ideally this question should be solve in 3 hours, at least a working version.

175. **You have a Smartphone class and will have derived classes like IPhone, AndroidPhone,WindowsMobilePhone can be even phone names with brand, how would you design this system of Classes.**
This is another design pattern exercise where you need to apply your object oriented design skill to come with a design which is flexible enough to support future products and stable enough to support changes in existing model.

176. **When do you overload a method in Java and when do you override it ?**

Rather a simple question for experienced designer in Java. if you see different implementation of a class has different way of doing certain thing than overriding is the way to go while overloading is doing same thing but with different input. method signature varies in case of overloading but not in case of overriding in java.

177. **Design ATM Machine ?**

We all use ATM (Automated Teller Machine) , Just think how will you design an ATM ? for designing financial system one must requirement is that they should work as expected in all situation. so no matter whether its power outage ATM should maintain **correct state (transactions**), think about **locking**, **transaction**, **error condition**, **boundary condition** etc. even if you not able to come up exact design but if you be able to point out non functional requirement, raise some question , think about boundary condition will be good progress.

178. **You are writing classes to provide Market Data and you know that you can switch to different vendors overtime like Reuters, wombat and may be even to direct exchange feed , how do you design your Market Data system.**

This is very interesting design interview question and actually asked in one of big investment bank and rather common scenario if you have been writing code in Java. Key point is you will have a MarketData interface which will have methods required by client e.g. getBid(), getPrice(), getLevel() etc and MarketData should be composed with a MarketDataProvider by using **dependency injection**. So when you change your MarketData provider Client won't get affected because they access method form MarketData interface or class.

179. **Why is access to non-static variables not allowed from static methods in Java**

You can not access non-static data from static context in Java simply because non-static variables are associated with a particular instance of object while Static is not associated with any instance. You can also see my post why non static variable are not accessible in static context for more detailed discussion.

180. **Design a Concurrent Rule pipeline in Java?**

Concurrent programming or concurrent design is very hot now days to leverage power of ever increasing cores in

advanced processor and Java being a multi-threaded language has benefit over others. Do design a concurrent system key point to note is thread-safety, immutability, local variables and avoid using static or instance variables. you just to think that one class can be executed by multiple thread a same time, So best approach is that every thread work on its own data, doesn't interfere on other data and have minimal synchronization preferred at start of pipeline. This question can lead from initial discussion to full coding of classes and interface but if you remember key points and issues around concurrency e.g. race condition, deadlock, memory interference, atomicity, ThreadLocal variables  etc you can get around it.

## Design pattern interview questions for Beginners

These software design and design pattern questions are mostly asked at beginners level and just informative purpose that how much candidate is familiar with design patterns like does he know **what is a design pattern** or **what does a particular design pattern do** ? These questions can easily be answered by memorizing the concept but still has value in terms of information and knowledge.

181. **What is design patterns ? Have you used any design pattern in your code ?**

Design patterns are tried and tested way to solve particular design issues by various programmers in the world. Design patterns are extension of code reuse.

182. **Can you name few design patterns used in standard JDK library?**

Decorator design pattern which is used in various Java IO classes, Singleton pattern which is used in Runtime , Calendar and various other classes, Factory pattern which is used along with various Immutable classes likes Boolean e.g. Boolean.valueOf and Observer pattern which is used in Swing and many event listener frameworks.

183. **What is Singleton design pattern in Java ? write code for thread-safe singleton in Java**

Singleton pattern focus on sharing of expensive object in whole system. Only one instance of a particular class is maintained in whole application which is shared by all modules. Java.lang.Runtime is a classical example of Singleton design pattern. You can also see my post 10 questions on Singleton pattern in Java for more questions and discussion. From Java 5 onwards you can use enum to thread-safe singleton.

184. **What is main benefit of using factory pattern ? Where do you use it?**

Factory pattern's main benefit is increased level of encapsulation while creating objects. If you use Factory to create object you can later replace original implementation of Products or classes with more advanced and high performance implementation without any change on client layer. See my post on Factory pattern for more detailed explanation and benefits.

185. **What is observer design pattern in Java**

Observer design pattern is based on communicating changes in state of object to observers so that they can take there action. Simple example is a weather system where change in weather must be reflected in Views to show to public. Here weather object is Subject while different views are Observers. Look on this article for complete example of Observer pattern in Java.

186. **Give example of decorator design pattern in Java ? Does it operate on object level or class level ?**

Decorator pattern enhances capability of individual object. Java IO uses decorator pattern extensively and classical example is Buffered classes like BufferedReader and BufferedWriter which enhances Reader and Writer objects to perform Buffer level reading and writing for improved performance. Read more on Decorator design pattern and Java

187. **What is MVC design pattern ? Give one example of MVC design pattern ?**

188. **What is FrontController design pattern in Java ? Give an example of front controller pattern ?**

189. **What is Chain of Responsibility design pattern ?**

190. **What is Adapter design pattern ? Give examples of adapter design pattern in Java?**

These are left for your exercise, try finding out answers of these design pattern questions as part of your preparation.

191. **Difference between constructors and static constructors?**

C++ no static constructor.

C# and Java Static constructors exist

They are used to initialize static members of a class.

The runtime executes them before the class is first used

To get the equivalent of a static constructor, you need to write a separate

1. **ordinary class to hold the static data and then make a**
2. **static instance of that ordinary class.**

```
class has_static_constructor {
    friend class constructor;

    struct constructor {
        constructor() { /* do some constructing here … */ }
    };

    static constructor cons;
};

// C++ needs to define static members externally.
has_static_constructor::constructor has_static_constructor::cons;
```

## Commonly Asked C++ Interview Questions | Set 1 - GeeksQuiz

192. **What are the differences between C and C++?**

1) C++ is a kind of superset of C, most of C programs except few exceptions (See this and this) work in C++ as well.

2) C is a procedural programming language, but C++ supports both procedural and Object Oriented programming.

3) Since C++ supports object oriented programming, it supports features like function overloading, templates, inheritance, virtual functions, friend functions. These features are absent in C.

4) C++ supports exception handling at language level, in C exception handling is done in traditional if-else style.

5) C++ supports references, C doesn't.

6) In C, scanf() and printf() are mainly used input/output. C++ mainly uses streams to perform input and output operations. cin is standard input stream and cout is standard output stream.

193. **What are the differences between references and pointers?**

194. **Both references and pointers can be used to change local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.**

195. **Despite above similarities, there are following differences between references and pointers.**

### References are less powerful than pointers

1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.

2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.

3) A reference must be initialized when declared. There is no such restriction with pointers

Due to the above limitations, references in C++ cannot be used for implementing data structures like

Linked List, Tree, etc. In Java, references don't have above restrictions, and can be used to implement all data structures. References being more powerful in Java, is the main reason Java doesn't need pointers.

## References are safer and easier to use:

1) Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise )

2) Easier to use: References don't need dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator (->) is needed to access members.

196.**Following things are necessary to write a C++ program with runtime polymorphism (use of virtual functions)**

1) A base class and a derived class.

2) A function with same name in base class and derived class.

3) A pointer or reference of base class type pointing or referring to an object of derived class.

For example, in the following program bp is a pointer of type Base, but a call to bp->show() calls show() function of Derived class, because bp points to an object of Derived class.

```
#include<iostream>
using namespace std;
class Base {
public:
    virtual void show() { cout<<" In Base \n"; }
};
class Derived: public Base {
public:
    void show() { cout<<"In Derived \n"; }
};
int main(void) {
    Base *bp = new Derived;
    bp->show();
    return 0;
}
```

Output:

```
In Derived
```

197.**What is this pointer?**

The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is a constant pointer that holds the memory address of the current object. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

198.**Can we do "delete this"?**

Ideally *delete* operator should not be used for *this* pointer. However, if used, then following points must be considered.

1) *delete* operator works only for objects allocated using operator *new* (See http://geeksforgeeks.org/?p=8539). If the object is created using new, then we can do *delete this*, otherwise behavior is undefined.

```
class A
{
  public:
    void fun()
    {
        delete this;
      }
};

int main()
{
  /* Following is Valid */
  A *ptr = new A;
  ptr->fun();
  ptr = NULL // make ptr NULL to make sure that things are not accessed
using ptr.


  /* And following is Invalid: Undefined Behavior */
  A a;
  a.fun();

  getchar();
  return 0;
}
```

2) Once *delete this* is done, any member of the deleted object should not be accessed after deletion.

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

class A
{
  int x;
  public:
    A() { x = 0;}
    void fun() {
      delete this;

      /* Invalid: Undefined Behavior */
      cout<<x;
    }
};
```

**The best thing is to not do *delete this* at all.**

199.**What are VTABLE and VPTR?**

vtable is a table of function pointers. It is maintained per class.

vptr is a pointer to vtable. It is maintained per object (See this for an example).

Compiler adds additional code at two places to maintain and use vtable and vptr.

1) Code in every constructor. This code sets vptr of the object being created. This code sets vptr to point to vtable of the class.

2) Code with polymorphic function call (e.g. bp->show() in above code). Wherever a polymorphic call is made, compiler inserts code to first look for vptr using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once vptr is fetched, vtable of derived class can be accessed. Using vtable, address of derived derived class function show() is accessed and called.

## Initialize a variable

- Traditionally
  - int i = 10;
- Constructor notation
  - int i(10);

```
Coversion/Coercion
= Type coversion, dynamic_cast, const_cast, static_cast
= Promotion,

Explicit   use as keyword
explicit B (const A& x) {} class conversion
foo = bar;       // calls type-cast operator

Implicit
pd = dynamic_cast<Derived*>(pba);
D& another_d = static_cast<D&>(br); // downcast
```

200. **What is a Inline Functions**

commonly used with classes.

If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

```
inline int Max(int x, int y)
```

201. **What is a translation unit**

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)

202. internal linking and external linking

static = linked internally when it can be accessed only from with-in the scope of a single translation unit.

extern= external linking a symbol can be accessed from other translation units as well

203. storage classes

used to specify the visibility/scope and life time of symbols(functions and variables).

specify where all a variable or function can be accessed and till what time those variables will be available during the execution of program.

204. **Reference Variable**

allows you create an alias (second name) for an already existing variable. A reference variable can be used to access (read/write) the original data.

```
int   a;
int&  b = a;
a = 10;

Value of a is 10 and value of Reference(b) is 10
a changes then b changes, both point to same address
```

205. **Whats the difference between a Reference Variable and a Pointer?**

-

| **Pointers** | Reference Variables |
|---|---|
| Pointers can be assigned to NULL | References **cannot be assigned NULL**. It should always be associated with actual memory, not NULL. |
| Pointers can be (**re)pointed to any object,** at any time, any number of times during the execution. | Reference variables should be initialized with an object when they are created and they **cannot be reinitialized** to **refer to another object** |
| Pointer has own memory address and location on stack | Reference variables has location on stack, but **shares the same memory location** with the object it refer to. |

206.**Is Encapsulation a Security device?**

No. Encapsulation is about error prevention. Security is about preventing purposeful attacks.

207.**What's the difference between the keywords struct and class?**

By default, `struct` members and base classes are `public`. With `class`, the default is `private`. Never rely on these defaults! Otherwise, `class` and `struct` behave identically.

208. **What's the big deal with generic programming?**

Generic programming" in the context of C++ refers to templates.

209.**Are virtual functions (dynamic binding) central to OO/C++?**

Sure, that's what makes C++ an object-oriented language. Don't switch from C to C++ unless you need virtual functions!

210.**What is a class?**

In OO software, "the fundamental building block".

211.**What's the deal with inline functions?**

Inlining a function call means that the compiler inserts the code of the function into the calling code (which is technically different, but logically similar to the expansion of `#define` macros). This may improve performance, because the compiler optimizes the callee code in the context of the calling code instead of implementing a function call. However, the performance impact depends on lots of things.

There's more than one way to say that a function should be inline, some of which use the `inline` keyword and some don't. No matter what way you use, the compiler might actually inline the function and it might not - you're just giving it a "hint". Sounds vague? It is - and it is *good*: it lets the compiler generate better and/or more debuggable code.

212.**class**

a type - a representation for a set of states (much like a C `struct`) and a set of operations for changing the state (moving from one state to another).

213.**What's a simple example of procedural integration?**

**FAQ:** There's an example of a function calling another function and how inlining may save you copying the parameters when you pass them to a function and copying the result it returns and stuff. There's also a disclaimer saying that it's just an example and many different things can happen.

214.**Do inline functions improve performance?**

**FAQ:** Sometimes they do, sometimes they don't. There's no simple answer.

215. **How can inline functions help with the tradeoff of safety vs. speed?**

**FAQ:** "In straight C" you could implement encapsulation using a `void*`, so that users can't access the underlying data. Instead, the users have to call functions which access that data by casting the `void*` to the right type first.

Not only is this type-unsafe - it's also costly, since the simplest access now involves a function call. In C++ you can use `inline` accessors to `private` data - safe, fast.

216. **Why should I use inline functions instead of plain old #define macros?**

**FAQ:** Because macros are evil. In particular, when a macro is expanded, the parameters are not evaluated before the expansion, but copied "as is" (they are interpreted as character sequences by the preprocessor, not as C++ expressions). Therefore, if a parameter is an expression with a side effect, such as `i++`, and the macro mentions it several times, the macro will expand to buggy code (for instance, `i` will get incremented many times). Or the generated code may be slow (when you use expressions like `functionTakingAgesToCompute()` as macro arguments).

Besides, inline functions check the argument types.

217. **What happens if you return a reference?**

**FAQ:** You can assign to the return value of a function. This is useful for operator overloading, as in `array[index] = value;` where array is an object of a class with overloaded `operator[]` returning a reference.

218. **How do you tell the compiler to make a non-member function inline?**

**FAQ:** Prepend the `inline` keyword to its prototype, and place the code in a header file, unless it's only used in a single `.cpp` file, or else you'll get errors about "unresolved externals" from the linker.

219. **Is there another way to tell the compiler to make a member function inline?**

**FAQ:** Yes, by writing its code right in the body of the class, instead of only writing the declaration there, and defining it outside of the class. This way, you don't even have to use the `inline` keyword.

It's easier when you write classes, but harder when you read them, because the interface is mixed with the implementation. Remember the "reuse-oriented world"? Think about the welfare of the many, many users of your class!

220. **What does object.method1().method2() mean?**

**FAQ:** This is called method chaining. `method1` returns a reference to an object of a class having `method2`. This is used in iostream - `cout << a << endl` is method chaining (the method is `operator<<`). A "slick" way to use method chaining is to simulate "named parameters": `a.setWidth(x).setHeight(y)`).

**FQA:** It's just like `object->method1()->method2()`, but with references instead of pointers.

221. **When should I use references, and when should I use pointers?**

**FAQ:** Use references unless you can't, especially in interfaces. In particular, references can't point to `NULL`, so you can't have a "sentinel reference". C programmers may dislike the fact that you can't tell whether a modified value is local or a reference to something else. But this is a form of *information hiding*, which is good because you should program in the language of a problem rather than the machine.

**FQA:** As with most duplicate features, there's no good answer to this question.

222. **What is a handle to an object? Is it a pointer? Is it a reference? Is it a pointer-to-a-pointer? What is it?**

**FAQ:** A "handle" is something identifying and giving access to an object. The term is meant to be vague, omitting implementation details (a handle can be a pointer or an index into an array or a database key, etc.). Handles are often encapsulated in smart pointer classes.

**FQA:** One very common way to implement handles in C relies on incomplete types and typedefs:

```
typedef struct FooState* Foo;
```

223. **Is there any difference between List x; and List x();?**

**FAQ:** Yes, and it's a *big* one. The first statement declares an object of type `List`, the second declares a function returning an object of type `List`.

**FQA:** There sure is quite a semantic difference. Too bad it's not accompanied by an equally noticeable syntactic difference. Which is why the question became a frequently asked one.

224. **Can one constructor of a class call another constructor of the same class to initialize the this object?**

**FAQ:** No. But you can factor out the common code into a private function. In some cases you can use default parameters to "merge" constructors. Calling placement `new` inside a constructor is very bad, although it can "seem" to do the job in some cases.

225. **Is the default constructor for Fred always Fred::Fred()?**

**FAQ:** No, the default constructor is the one that can be called without arguments - either because it takes none or because it defines default values for them.

226. **Which constructor gets called when I create an array of Fred objects?**

**FAQ:** The default constructor. If there isn't any, your code won't compile. But if you use `std::vector`, you can call any constructor. And you should use `std::vector` anyway since arrays are evil. But sometimes you need them. In that case, you can initialize each element as in `Fred arr[2] = {Fred(3,4), Fred(3,4)};` if you need to call constructors other than the default. And finally you can use placement new - it will take ugly declarations and casts, you should be careful to align the storage right (which can't be done portably), and it's hard to make this exception-safe. See how evil those arrays are? Use `std::vector` - it gets all this complicated stuff right.

227. **hould my constructors use "initialization lists" or "assignment"?**

**FAQ:** Initialization lists - that's more efficient, except for built-in types (with initializers, you avoid the construction of an "empty" object and the overhead of cleaning it up at the assignment). And some things can't be initialized using assignment, like member references and const members and things without default constructors. So it's best to always use initialization lists even if you don't have to for the sake of "symmetry". There are exceptions to this rule. There's no need for an exhaustive list of them - search your feelings.

228. **Should you use the this pointer in the constructor?**

**FAQ:** Some people think you can't since the object is not fully initialized, but you can if you know the rules. For example, you can always access members inside the constructor body (after the opening brace). But you can never access members of a derived class by calling virtual functions (the function call will invoke the implementation of the class defining the constructor, not that of the derived class). Sometimes you can use a member to initialize another member in an initializer list and sometimes you can't - you must know the order in which members are initialized.

229. **Does return-by-value mean extra copies and extra overhead?**

**FAQ:** "Not necessarily". A truly exhausting, though not necessarily exhaustive list of examples follows, with many stories about "virtually all commercial-grade compilers" doing clever things.

230. **Why can't I initialize my static member data in my constructor's initialization list?**

**FAQ:** Because you must define such data explicitly as in `static MyClass::g_myNum = 5;`.

231. **Why are classes with static data members getting linker errors?**

**FAQ:** Because you must define such data explicitly as in `static MyClass::g_myNum = 5;`.

FQA: Beyond being annoying, this is quite weird. At the first glance it looks reasonable: after all, C++ is just a thick layer of syntax on top of C, but the basic simpleton mechanisms are the same. For instance, definition look-up is still done using header files holding random declarations and object files holding arbitrary definitions - each function can be declared anywhere (N times) and defined anywhere (1 time).

So the compiler can't let you define something which becomes a global variable at the C/assembly level in a header file as in `static int g_myNum = 5;` - that way, you'd get multiple definitions (at each file where the class definition is included). Consequently, the C++ syntax is defined in a way forcing you to solve the compiler's problem by choosing a source file and stuffing the definition there (most frequently the choice is trivial since a class is implemented in a single source file, but this is a *convention*, not a rule the compiler can use to simplify definition look-up).

232. **What's the "static initialization order fiasco"?**

**FAQ:** A subtle, frequently misunderstood source of errors, which are hard to catch because they occur before `main` is called. The errors can happen when the constructor of a global object defined in `x.cpp` accesses (directly or indirectly) a global object defined in `y.cpp`. The order of initialization of these objects is undefined, so you'll see the problem in 50% of the cases (an error may be triggered by a rebuild). "It's that simple".

233. **How do I prevent the "static initialization order fiasco"?**

**FAQ:** By using the "Construct On First Use" idiom, as in

```
MyClass& getMyObj()
{
  static MyClass* p = new MyClass;
  return *p;
```

234. **Why am I getting an error after declaring a Foo object via Foo x(Bar())?**

**FAQ:** This *hurts*. *Sit down*.

The "simple" explanation: this doesn't really declare an object; `Foo x = Foo(Bar());` does. The complete explanation for "those caring about their professional future": this actually declares a function named `x` returning a `Foo`; the single argument of this `x` function is of type "a function with no argument returning a `Bar`".

235. **What's the order that local objects are destructed?**

**FAQ:** In reverse order of construction - the stuff declared last is destroyed first.

236. **Can I overload the destructor for my class?**

**FAQ:** No. Destructors never have parameters or return values. And you're not supposed to call destructors explicitly, so you couldn't use parameters or return values anyway.

237. **What if I want a local to "die" before the close } of the scope in which it was created? Can I call a destructor on a local if I *really* want to?**

**FAQ:** No, no, no! See the next FAQ for a simple solution. But don't call the destructor!

**FQA:** If you ever get into the situation of promoting an especially disgusting product, such as the C++ programming language, there are better ways to handle it than get all excited. You can try and find a less disgusting product to center your money-making around, or you can just relax.

"Don't call the destructor". There has to be *some* reason for the destructor call to compile, doesn't it? Perhaps sharing it with us could help calm down.

238. **What is "placement new" and why would I use it?**

**FAQ:** There are many uses for placement `new`. For example, you can allocate an object in a particular location, you can pass the pointer to this location to placement `new` like this: `C* p = new(place) C(args);`

Don't use this unless you really care where the object lives (say, you have a memory-mapped hardware device like a timer, and you want to have a `Timer` object at the particular location).

Beware! It is your job to make sure that the address where you allocate the object is properly aligned and that you have enough space for the object. You are also responsible for calling the destructor with `p->~C();`.

239. **What is "self assignment"?**

**FAQ:** It's assigning the object to itself, directly or indirectly. For example:

```
void f(C& x, C& y) { x=y; }

void g(C& x) { f(x,x); }
```

240. **What are the benefits of operator overloading?**

**FAQ:** You can "exploit the intuition" of the users of your classes. Their code will speak in the language of the problem instead of the language of the machine. They'll learn faster and make less errors.

**FQA:** The keyword in the FAQ's answer is "exploit". Your intuition tells you that `a+b` just works - and in the "problem domain", you are right. But in C++, it doesn't take the machine too long to raise its ugly iron head and start talking to you in its barbaric language, and it has all the means to make you listen.

241. **What are some examples of operator overloading?**

**FAQ:** Here are a few (some real, some hypothetical), there are many more.

`str1 + str2` concatenates a couple of `std::string` objects.
`NapoleonsBirthday++` increments an object of class `Date`.
`a*b` multiplies two `Numbers`.
`a[i]` accesses the i'th element of a user-defined `Array` class object.
`x = *p` dereferences a "smart pointer" which actually represents an address of a disk record; the dereferencing seeks to that location on the disk and fetches the record into `x`.

242. **What operators can/cannot be overloaded?**

**FAQ:** Most can be overloaded. You can't overload the C operators `.` and `?:` (and `sizeof`). And you can't overload the C++ operators `::` and `.*`. You can overload everything else.

243. **Which is more efficient: `i++` or `++i`?**

**FAQ:** It's the same for built-in types. `++i` may be faster for user-defined types, because there's no need to create a copy that the compiler may fail to optimize out. Most likely the overhead is small, but why not pick the habit of using `++i`?

244. **What are some advantages/disadvantages of using `friend` functions?**

**FAQ:** Advantage: they allow the designer to choose between `obj.func()` and `func(obj)`, which "lowers maintenance costs".

Disadvantage: they require more code to achieve dynamic binding. Non-member functions can't be `virtual`, so if the designer's syntax of choice is `func(obj)`, and dynamic binding is needed, `func` will have to delegate to a `protected virtual` member with `obj.func()`.

245. **Why should I use `<iostream>` instead of the traditional `<cstdio>`?**

**FAQ:** There are four reasons:

*Increase type safety*: with `iostream`, the compiler knows the types of the things you print. `stdio` only figures them out at run time from the format string.

*Reduce the number of errors*: with `stdio`, the types of objects you pass must be consistent with the format string; `iostream` removes this redundancy - there is no format string, so you can't make these errors.

*Printing objects of user-defined types*: with `iostream`, you can overload the operators `<<` and `>>` to support new types, and the old code won't break. `stdio` won't let you extend the format string syntax, and there seems to be no way to support this kind of thing in a way avoiding conflicts between different extensions.

*Printing to streams of user-defined types*: you can implement your own stream classes by deriving from the base classes provided by `iostream`. `FILE*` can not be extended because "it's not a real class".

246. **How does that funky `while (std::cin >> foo)` syntax work?**

**FAQ:** `istream` has overloaded `operator void*`. The compiler calls this operator in boolean contexts (when it expects a condition, for example), because `void*` can be converted to a boolean. The operator returns `NULL` when there's nothing left to read, or when a format error occurred previously.

247. **Why does my input seem to process past the end of file?**

**FAQ:** Because the stream doesn't know that you've reached the end of file until you actually try to read past the end of file. For instance, a stream object may read characters which are entered interactively using a keyboard. So it's impossible to tell when it's over.

248. **Should I end my output lines with `std::endl` or `'\n'`?**

**FAQ:** The former has the additional side-effect of flushing the output buffer. Therefore, the latter will probably work faster.

249. **How can I provide printing for an entire hierarchy of classes?**

**FAQ:** You can create a `protected virtual` method that each class will override to do the printing, and call it from a `friend operator<<`.

**FQA:** Fantastic, except for the `protected` and the `friend operator<<` part. Of course you can use a plain `public virtual` method instead.

250. **How can I write/read objects of my class to/from a data file?**

**FAQ:** Read the section about serialization.

**FQA:** The FQA doesn't have a section about a serialization. Short summary: you're in for a rude awakening. There's no standard serialization mechanism in C++. Furthermore, there's no way to define a reasonable custom one since there's no reflection (no way to figure out the structure of an arbitrary object given a pointer to it, no way to create an object of a class given its name or some other sort of ID, etc.).

However, there are many custom packages for serialization (typically thousands of source code lines, requiring you to use hairy macros/templates for each serialized class member). Or you can roll your own, or you can dump the whole snapshot of your process to a file in non-portable ways (may be the cheapest thing to do more frequently than it sounds).

251. **Why can't I open a file in a different directory such as `"..\test.dat"`?**

**FAQ:** Because `\t` expands to the tab character. Use `\\t` to say "backslash followed by t".

252. **How can I tell {if a key, which key} was pressed before the user presses the ENTER key?**

**FAQ:** The C++ standard doesn't even assume your system has a keyboard. So there's no portable way.

253. **Does `delete p` delete the pointer `p`, or the pointed-to-data `*p`?**

**FAQ:** That would be `*p`. The keyword should have been `delete_whatever_is_pointed_by`. Similarly, `free` should have been called `free_whatever_is_pointed_by`.

254. **Why should I use `new` instead of trustworthy old `malloc()`?**

**FAQ:** `new/delete` call the constructor/destructor; `new` is type safe, `malloc` is not; `new` can be overridden by a class.

255. **In `p = new Fred()`, does the `Fred` memory "leak" if the `Fred` constructor throws an exception?**

**FAQ:** No, because the compiler effectively generates a `try/catch` block around the constructor call, and when exceptions are thrown, the `catch` part deallocates the memory and rethrows the exception.

**FQA:** If you throw exceptions in constructors, make sure they are caught when custom allocators are used (the kind that looks like this: `new (pool.alloc()) Fred()`).

If you don't throw exceptions, the implicit `try/catch` around `new` is one illustration of the fact that exceptions increase the size of your code even when you don't use them. One good thing is that most compilers have a flag disabling exceptions.

256. **Does C++ have arrays whose length can be specified at run-time?**

**FAQ:** That would be yes - there's `std::vector`.

Actually, it's a no - you can't do it with built-in arrays.

Wait a minute - you *can* do it with *the first index* of a built-in array, for example: `new T[run_time_val][compile_time_val]`.

You know what? Arrays are evil.

257. **What are the two kinds of garbage collectors for C++?**

**FAQ:** There are *conservative* and *hybrid* garbage collectors. The conservative ones just look for bit sequences looking like pointers. The hybrid ones require the programmer to specify some layout information explicitly in the code (but they still traverse the call stack conservatively when they look for pointers).

Garbage collectors may cause memory leaks when a bit pattern looking like a pointer is misinterpreted as a proof that the block pointed by this "pointer" is still in use. And some illegal programs may "confuse" garbage collectors (that's the word used by the FAQ) by keeping pointers outside of allocated blocks. Why can't these programmers behave reasonably?

258. **What is "`const` correctness"?**

**FAQ:** Oh, that's a great thing. You declare an object as `const` and prevent it from being modified.

For example, if you pass a `std::string` object to a function by `const` pointer or reference, the function won't modify it and it won't be copied the way it happens when the object is passed by value.

259. **How is "`const` correctness" related to ordinary type safety?**

**FAQ:** It's one form of type safety. You can think about `const std::string` as an *almost* separate type that lacks certain operations of a string, like assignment. If you find type safety useful, you'll also like `const` correctness.

260. **Should I try to get things `const` correct "sooner" or "later"?**

**FAQ:** As soon as possible, because when you add `const` to a piece of code, it triggers changes in every place related to it.

261. **What does "`const Fred* p`" mean?**

**FAQ:** A pointer to a `const Fred` object. The object can't be changed, for example, you can't call methods not qualified as `const`.

262. **What's the difference between "`const Fred* p`", "`Fred* const p`" and "`const Fred* const p`"?**

**FAQ:** In the first example, the object is immutable. In the second example, the pointer is immutable. In the third example, both are immutable.

**FQA:** Um, right. Remember: `const` makes your programs readable.

263. **What does "`const Fred& x`" mean?**

**FAQ:** It's a reference to an immutable Fred object.

FQA: Which is similar to a pointer to an immutable Fred object. However, the FAQ holds the "references are NOT pointers" religion (specifically, it belongs to the "pointers are evil" faction), so it dutifully replicates the explanation given in the answer about the pointer case, replacing "pointer" with "reference".

264. **Does "`Fred& const x`" make any sense?**

**FAQ:** No. It says that you can change the object, but not the reference. But you can never change a reference anyway, it will always refer to a single object.

265. **What does "`Fred const& x`" mean?**

**FAQ:** It's equivalent to "`const Fred& x`". The question is - which form should you use? Nobody can answer this for *your* organization without understanding *your* needs. The are lots of business scenarios, some producing the need for one form, others for the other. The discussion takes a full screen of text.

266. **What does "`Fred const* x`" mean?**

**FAQ:** The FAQ replicates the previous answer, replacing "reference" with "pointer". Probably the same religion thing again.

267. **What is a "`const` member function"?**

**FAQ:** It's declared by appending `const` to the prototype of a class member function. Only this kind of methods may be called when you have a `const` object. Errors are caught at compile time, without any speed or space penalty.

268. **What's the relationship between a return-by-reference and a `const` member function?**

**FAQ:** `const` member functions returning references to class members should use a `const` reference. Many times the compiler will catch you when you violate this rule. Sometimes it won't. You have to think.

269. **What's the deal with "`const`-overloading"?**

**FAQ:** You can have two member functions with a single difference in the prototype - the trailing `const`. The compiler will select the function based on whether the `this` argument is `const` or not. For instance, a class with a subscript operator may have two versions: one returning mutable objects and one returning `const` objects - in the latter version, `this` is also qualified as `const`.

270. **Does `const_cast` mean lost optimization opportunities?**

**FAQ:** Theoretically, yes. In practice, no. If a compiler knows that a value used in a piece of code is not modified, it can optimize the access to that value, for example, fetch it to a machine register. However, the compiler can't be sure that a value pointed by a `const` pointer is not modified because of the aliasing problem: there can be other pointers to the same object, not necessarily constant. Proving the opposite is almost always impossible.

And when the compiler can't prove it, it can't speed up the access to the value. For example, if it uses the value it fetched to a register after someone modified it in the original memory location, the compiler changes the meaning of the program because it uses an outdated value.

271. **Why does the compiler allow me to change an `int` after I've pointed at it with a `const int*`?**

**FAQ:** Because when you point to something with a `const` pointer, this only means that you can't use that pointer to change the object. It doesn't mean the object can't be changed at all, because it can be accessible in other ways, not only through this pointer.

272. **Why am I getting an error converting a `Foo**` to `const Foo**`?**

**FAQ:** It would be "invalid and dangerous"! Suppose `Foo**` points to an array of pointers, which can be used to modify the pointed objects. Suppose C++ would allow to pass this array to a function which expects `const Foo**` - an array of pointers which *can't* be used to modify the pointed objects.

Now suppose that this function fills the array with a bunch of pointers to *constant* objects. This looks perfectly good in that context, because that's what the function was passed - an array of pointers to constant objects. But what we've got now is an array of pointers which can be used to modify those `const` objects, because the declaration of the array does allow such modifications!

It's a good thing we get a compile time error instead. Don't use casts to work around this!

273. **How can I protect derived classes from breaking when I change the internal parts of the base class?**

**FAQ:** Your class has two interfaces: `public` and `protected`. To protect your users, don't expose data members. Similarly, you can protect the derived classes by not having `protected` data members; provide `protected inline` accessors instead.

**FQA:** Dear Design Guru! Listen carefully, and try to understand.

One. Writing get and set function for every member takes time, and so does reading them. This time can instead be used to get something done.

Two. If you can get and set a member, it's pretty close to being public. Any non-trivial representation change becomes impossible since the ability to set this particular member is now a part of the contract.

Three. Having properties in the language - things that look like members but are in fact accessors - doesn't hurt. Many languages have it, allowing to use plain members and in the 0.1% of the cases when your class becomes very popular *and* you want to change it *and* you can do it without breaking the contract, you can make the member a property. There's no reason for not having this in C++ that's even remotely interesting to a language user.

274. **I've been told to never use protected data, and instead to always use private data with**

**protected access functions. Is that a good rule?**

**FAQ:** No, no, no - "always" rules don't work in the real world! You don't have time to make life easy for everyone. Spend your time making it easy for the important people. People are not equal. The important people are called "customers".

275.**What is a "`virtual` member function"?**

**FAQ:** The most important thing about C++ if you look from an OO perspective.

With `virtual` functions, derived classes can provide new implementations of functions from their base classes. When someone calls a `virtual` function of an object of the derived class, this new implementation is called, even if the caller uses a pointer to the base class, and doesn't even know about the particular derived class.

The derived class can completely "override" the implementation or "augment" it (by explicitly calling the base class implementation in addition to the new things it does).

276.**How can C++ achieve dynamic binding yet also static typing?**

**FAQ:** *Static typing* means that when a function (`virtual` or other) is called, the compiler checks that the function call is valid according to the statically defined interfaces.

*Dynamic binding* means that the code generated from the statically checked function calls may actually call many different implementations, and figures out the one to call at run time.

Basically C++ allows many things to be done upon a `virtual` function call, as long as these things follow a specified protocol - the base class definition.

277.**What's the difference between how `virtual` and non-`virtual` member functions are called?**

**FAQ:** non-`virtual` functions are resolved at compile time. `virtual` functions are resolved at run time.

The compiler must generate some code to do the run-time resolution. This code must be able to look at an object and find the version of the function defined by the class of the object. This is usually done by creating a table of virtual functions for each class having them (the "vtable"). All objects of the class have an implicitly generated member pointer (the "vptr"), initialized to point to the class vtable by all constructors.

To implement a virtual function call, the compiler generates code similar to that it would produce from the C expression `(*p->vptr->func_ptr)`. That is, it dereferences the object pointer to fetch the vptr, then fetches the function pointer from a fixed offset, then calls the function through the pointer.

The exact cost depends on complicated stuff like page faults, multiple inheritance, etc.

278.**How can a member function in my derived class call the same function from its base class?**

**FAQ:** For some a reason, a pretty low-level discussion follows. Name mangling, "call-by-name" vs "call-by-slot-number", code listings with double underscores...

**FQA:** In a `class Derived`, in a function `Derived::f()`, you can type `Base::f();` to call the `f` implementation from your base class `Base`. The compiler will ignore the actual type of your object (at the low level - vtable and all that), and call `Base::f` just the way non-`virtual` functions are normally called.

I think you can do that in every OO language. It's pretty natural.

279.**What is a "`virtual` constructor"?**

**FAQ:** It's an idiom allowing you to have a pointer to a base class and use it to create objects of the right derived class. You can implement it by providing `virtual` functions like these:

```
virtual Base* create() const;
virtual Base* copy() const;
```

You implement them like this:

```
Derived* Derived::create() const { return new Derived; }
Derived* Derived::copy() const { return new Derived(*this); }
```

Note that we return `Derived*`, not `Base*` - it's called "Covariant Return Types". Some compilers have it, some don't.

280. **When should my destructor be `virtual`?**

**FAQ:** The rule of thumb is - when you have a `virtual` function. Strictly speaking, you need it when you want someone to be able to derive classes from your class, create objects with `new`, and `delete` them via pointers to a base class.

**FQA:** The situations when the rule of thumb is not good enough were not reported on our planet. Use this rule of thumb, if only to suppress compiler warnings. Too bad the C++ compiler doesn't use the rule silently itself - it could simply make the destructor `virtual` in these cases.

281. **Should I hide member functions that were public in my base class?**

**FAQ:** No, no, don't even think about it, *don't do that*, no. Your desire is probably the result of "muddy thinking".

**FQA:** With all due respect, it is your precious programming language that probably is the result of "muddy thinking". The question talks about overriding base class functions in the `private` section of your derived class. This is trivially and reliably detectable at compile time. If you get so excited about how wrong it is, *why does it compile*?

282. **What's the big deal of separating interface from implementation?**

**FAQ:** Interfaces are the most important thing possessed by a company. Defining interfaces is hard, throwing together an implementation is easy. Designing interfaces takes lots of time and is done by people who are paid lots of money.

You wouldn't want these valuable artifacts clobbered by implementation details, would you?

283. **How do I separate interface from implementation in C++ (like Modula-2)?**

**FAQ:** Use an abstract base class.

284. **What is an ABC?**

**FAQ:** An abstract base class.

Logically, it corresponds to an abstract concept, such as `Animal`. Specific animals, like `Rat`, `Skunk` and `Weasel`, correspond to implementation classes derived from `Animal`.

Technically, a class with at least one pure virtual function is abstract. You can't have objects of abstract classes, you can only create objects of classes derived from them that implement all the pure virtual functions.

285. **What is a "pure virtual" member function?**

**FAQ:** It's a virtual function which (normally) has no implementation in the base class, and (always) must be implemented in a derived class to make it non-abstract. The syntax is: `virtual result funcname(args) = 0;`.

This declaration makes the class abstract - it is illegal to create objects of this class. A derived class that doesn't implement at least one pure virtual function is still abstract. Only derived classes without any unimplemented pure virtual functions can be instantiated (you can create objects of these classes).

286. **Is it okay for a non-`virtual` function of the base class to call a `virtual` function?**

**FAQ:** Sometimes it is. Suppose you have a class `Animal` with a non-`virtual` `getAwfullyExcited()` method. In your system, all animals do it similarly:

```
void Animal::getAwfullyExcited()
{
  makeExcitedNoises(); // all animals make different noises
  cout << "And here comes the dance!" << endl; // all animals always warn people
before they dance
  danceExcitedly(); // all animals dance differently
}
```

So there you have it - an orderly hierarchy of animals, each getting excited according to the standard procedure in its own unique way.

287. **That last FAQ confuses me. Is it a different strategy from the other ways to use virtual functions? What's going on?**

**FAQ:** Yes, there are two different strategies related to the use of `virtual` functions. In the first case, you have a common non-`virtual` method in the base class, but use `virtual` methods to implement the parts which do differ. In the second case, you have `virtual` methods implemented differently in derived classes, but they call common non-`virtual` methods in the base class. These common bits can also be implemented somewhere else - not necessarily in the base class.

Sometimes you use both strategies in the same class - one for some of the methods, the other for other methods. That's OK, too.

288. **Should I use protected virtuals instead of public virtuals?**

**FAQ:** Well, first of all avoid strict rules saying "never do this, always do that". As a rule of thumb, experience tells that most of the time virtual functions are best made public, with two notable exceptions. First, there are functions which are supposed to be called only from the base class, the way described in the previous two FAQs. And second, there's the *Public Overloaded Non-Virtuals Call Protected Non-Overloaded Virtuals* Idiom:

```
class Base
{
public:
  void overloadingTotallyRules(int x)   {
butWeDontWantToOverloadVirtualFunctions_int(x); }
  void overloadingTotallyRules(short x) {
butWeDontWantToOverloadVirtualFunctions_short(x); }
protected:
  virtual void butWeDontWantToOverloadVirtualFunctions_int(int);
  virtual void butWeDontWantToOverloadVirtualFunctions_short(short);
};
```

This solves an important problem: overloading totally rules, but we don't want to overload virtual functions. The *Public Overloaded Non-Virtuals Call Protected Non-Overloaded Virtuals* Idiom makes life easy for class users *and* the implementors of derived classes! Clearly you, the author of the base class, absolutely *must* clobber your code with this nonsense to make everyone's life better! Think about the reduced maintenance costs and the benefit of the many, you selfish code grinder.

289. **When should someone use private virtuals?**

**FAQ:** Probably never. It confuses people, because they don't think `private` virtuals can't be overridden, but they can.

**FQA:** The FAQ is right - the behavior of `private virtual` is ridiculous; it compiles, but for the wrong reasons. Note that there are many, many more things in C++ that primarily confuse people. One excellent reason to avoid C++ altogether. I'm not joking. Nothing funny about it. Well, maybe it is funny that C++ developers are confused for a living, but that's a cruel kind of humor.

290. **When my base class's constructor calls a virtual function on its this object, why doesn't my derived class's override of that virtual function get invoked?**

**FAQ:** Suppose you have a base class called `Base`, calling a virtual function `f` in its constructor. Then, when objects of a derived class called `Derived` are created, `Base::virt` is called from `Base::Base`, not `Derived::f`.

The reason is that when `Base::Base` executes, the object is still of type `Base`. It only becomes an object of type `Derived` when the code in `Derived::Derived` is entered. If you wonder why C++ works this way, consider the fact that `Derived::f` could access uninitialized members of the class `Derived` if it could be called from `Base::Base`, which runs before `Derived::Derived` initializes the members of `Derived`.

Luckily, C++ doesn't let this happen, preventing subtle errors!

291. **Okay, but is there a way to *simulate* that behavior as *if* dynamic binding worked on the this**

**object within my base class's constructor?**

FAQ: Yes. It's an Idiom, of course. Namely, the *Dynamic Binding During Initialization Idiom*.

One option is to have a `virtual init` function to be called after construction. Another option is to have a second hierarchy of classes, which doesn't always work, but... (sorry, I couldn't make it through all the code listings. If you like lots of hierarchies of classes and find this solution interesting, please follow the link to the FAQ's answer).

The first approach has the problem of requiring an extra function call upon initialization. We can rely on the self-discipline of the programmers (the self-discipline is especially important when exceptions can be thrown by `init` - make sure you release the allocated object properly), or we can wrap the construction and the `init` call in a single `create` function returning a pointer to the object. The latter rules out allocation on the stack.

292. **I'm getting the same mess with destructors: calling a `virtual` on my `this` object from my base class's destructor ends up ignoring the override in the derived class; what's going on?**

FAQ: Again, you're being protected by the compiler. Protected from yourself! You could access members that were already destroyed if the compiler let you do what you want.

When `Base::~Base` is called, the type of the object is changed from `Derived` to `Base`.

**FQA:** Thanks for the protection. At least the behavior is symmetrical.

Protect me from access to destroyed objects through dangling references in *the general case* next time, will you?

293. **Should a derived class redefine ("override") a member function that is non-`virtual` in a base class?**

FAQ: You can do that, but you shouldn't.

Experienced programmers sometimes do that for various reasons. Remember that the user-visible effects of both versions of the functions must be identical.

294. **What does it mean that the "virtual table" is an unresolved external?**

FAQ: Well, as you know, "unresolved external" means that there's a function or a global variable that your code declares and uses, but never defines. A virtual table is a global variable declared implicitly by a typical C++ implementation for each class with at least one virtual function.

Now, normally when you forget to define a virtual function, you'll get an "unresolved external" error saying that this function is missing. But in many implementations, if you forget the *first* virtual function, the whole virtual table will become "unresolved". That's because these implementations define the virtual table at the translation unit where the first virtual function is implemented.

295. **How can I set up my class so it won't be inherited from?**

FAQ: You can have the constructors `private`. The objects will have to be created by `public` functions delegating to the constructor.

Alternatively, you can comment the fact that you don't want the class to be inherited from: `// if you inherit from this class, I'll hunt you down and kill you.`

There's a third solution (**WARNING** - this one can rot your brain): you can inherit your class from a `private virtual` base class, which has a `private` constructor and declares your class a `friend`. This way, if someone tries to inherit from you, they'll need to directly call the base class of the constructor, which won't compile, since the constructor is `private`. This can add an extra word of memory to the size of your objects though.

296. **How can I set up my member function so it won't be overridden in a derived class?**

FAQ: Use a `/* final */` comment instead of a `final` keyword. It's not a technical solution - so what? The important thing is that it works.

297. **Is the type of "pointer-to-member-function" different from "pointer-to-function"?**

FAQ: It is.

If you have a non-member function `void f(int)`, then `&f` is of type `void (*)(int)`.

If you have a non-`static` member function `void C::f(int)`, then `&C::f` is of type `void (C::*)(int)`.

298. **How do I pass a pointer-to-member-function to a signal handler, X event callback, system call that starts a thread/task, etc?**

FAQ: You don't. A member function can't be used without an object of the class, so the whole thing can't work. What you can do is write a non-member function wrapping your pointer-to-member-function call.

For example, thread creation callbacks usually have a `void*` argument. You could pass an object pointer in that argument to the callback (which has to be a non-member). The callback would then cast the `void*` down to the actual type and call the object's method.

Some functions, like `signal`, use callbacks without a `void*` argument or anything similar. In that case, you have no choice but save a pointer to the object in a global variable. The callback can get the object pointer from that global variable and call the method.

`static` member functions *can* be used in the contexts where a C callback is expected, if they are `extern "C"`. Although on most compilers it would probably work without `extern "C"`, the standard says it doesn't have to work.

299. **Can I convert a pointer-to-member-function to a void*?**

FAQ: No, and if it seems to work on some platform, it doesn't make it legal.

300. **I need something like function-pointers, but with more flexibility and/or thread-safety; is there another way?**

FAQ: A functionoid is what you need.

**FQA:** "Functionoid" rhymes with "marketroid", and is a term local to the FAQ, used instead of the standard term "functor".

301. **What the heck is a functionoid, and why would I use one?**

FAQ: It means "a function on steroids", of course. The FAQ goes on to describe a class with a single `virtual` function called `doit`. An object of this class is basically just like a function except you can pass it arguments in its constructor, and it can keep state between calls without thread-unsafe global variables. The discussion is very lengthy and didactic, there are lots of examples. If you didn't say to yourself something like "Oh, yeah, that problem", you can follow the link to the real FAQ's answer to see what this is all about.

302. **Can you make functionoids faster than normal function calls?**

FAQ: Sure! Instead of `virtual` functions, use `inline` member functions and make the code using the "functionoid" a template.

303. **What's the syntax / semantics for a "function template"?**

FAQ: Pretty similar to class templates, plus you can usually omit the template parameters - the compiler will figure them out from the function arguments. For instance, you can write a `swap` function for swapping two values of any type, be it integers, strings, sets or file systems (yes, the FAQ *actually mentions* swapping some `FileSystem` objects).

By the way, an instantiation of a "function template" is called "template function".

304. **How do I explicitly select which version of a function template should get called?**

FAQ: Most of the time you don't need to do it - the compiler will guess. It uses arguments to guess, so when your function has none, use `f<int>();`. Sometimes you want to force the compiler to choose a different type than it would choose by default - for example, `g(45)` will instantiate `g<int>`, while you want `g<long>`. You can force the compiler to call `g<long>` with explicit instantiation as in `g<long>(45);` or type conversion as in `g(45L);`.

305. **What is a "parameterized type"?**

FAQ: A way to say "class templates".

306. **What is "genericity"?**

FAQ: A way to say "class templates". Don't confuse with "generality".

307. **My template function does something special when the template type T is int or std::string; how do I write my template so it uses the special code when T is one of those specific types?**

FAQ: First, make sure it's a good thing to do in your case. Generally, it is when the "observable behavior" in the special version you want to add is identical to the general case - otherwise, you're not helping your users. If the special case is "consistent" with the generic case, you can do it as in `template<> void foo<int>() { ... }`.

308. **Huh? Can you provide an example of template specialization that doesn't use foo and bar?**
**FAQ:** For instance, you can "stringify" values of different types using a template. The generic version boils down to `ostringstream out; out << x`. But you might want to define specializations to handle types where the `ostream` output operator doesn't do what you like (you can set the output precision of floating point numbers, etc.)

309. **All those templates and template specializations must slow down my program, right?**
**FAQ:** You guessed wrong. Maybe the compilation will become "slightly" slower. But the compiler ends up figuring out the types of everything, and then doing all the usual nifty C++ optimizations.

310. **So templates are overloading, right?**
**FAQ:** They are in the sense that they are inspected when the compiler resolves names (figures out the version of `f` that should be called by `f(x)`). They are not in the sense that the rules are different. Specifically, there's the SFINAE (Substitution Failure Is Not An Error) rule: the argument types have to match exactly for a template to be considered in overload resolution. If they don't, the compiler won't try to apply conversions the way it would with a function - instead it will discard the template.

311. **Why can't I separate the definition of my templates class from its declaration and put it inside a .cpp file?**
**FAQ:** "Accept these facts", says the FAQ - templates are not code, just a recipe for generating code given parameters; in order to compile this code, it must first be generated, which takes knowing both the template definition and the definitions of the parameters which are passed to it; and the compiler doesn't know anything about code outside of a file when compiling the file (which is called "separate compilation").

So you have to place template definitions in header files, or else the compiler won't get a chance to see all the definitions it needs at the same time. Experts should calm down - yes, it's oversimplified; if you know it is, you don't need this answer anyway.

312. **How can I avoid linker errors with my template functions?**
**FAQ:** You probably didn't make the definition of a template available to your compiler at the point where a template is used (did you implement a template in a .cpp file?). There are three solutions:

- Move the definition to the .h file (which may increase the size of your compiled code, unless the compiler is "smart enough").
- Add explicit instantiations in your .cpp file. For example, `template void foo<int>();` will cause the compiler to generate the code of `foo<int>`, and the linker will find it.
- `#include` the .cpp file defining the template at the .cpp file using the template. If it feels weird, live with it or read the previous FAQ.

313. **How does the C++ keyword export help with template linker errors?**
**FAQ:** It's "designed" to eliminate the need to make the definition of a template available to the compiler at the point of usage. Currently, there's only one compiler supporting it. The keyword's future is "unknown".

An advice for futuristic programmers follows. It shows a way to make code compatible with both compilers that support `export` and those that don't using - guess what? - the wonders of the evil C preprocessor. Among other things, the FAQ advises to `#define export` under certain conditions.

314. **Why am I getting errors when my template-derived-class uses a nested type it inherits from its template-base-class?**
**FAQ:** This can hurt, sit down. The compiler doesn't look for "non-dependent" names (ones that don't mention the template parameters) in "dependent" base classes. So if you inherited a nested class or typedef `A` from your base class `B<T>`, you can only access it using a dependent name, like `B<T>::A`, but you can't use a non-dependent name, like plain `A`.

*And* you'll have to prefix that "dependent" name with the `typename` keyword. That's because the compiler doesn't know that `B<T>::A` is a type (think about two specializations, one defining a nested class `A` and one defining a global variable `A`).