C++ Interview Questions

Compiled by Dr. Fatih Kocan, Wael Kdouh, and Kathryn Patterson

for the Data Structures in C++ course(CSE 3358)

Spring 2008

Contents

Constructors/Destructors

Q: Is it possible to have Virtual Constructor? If yes, how? If not, Why not possible?
A: There is nothing like Virtual Constructor. The Constructor can't be virtual as the constructor is a code which is responsible for creating an instance of a class and it can't be delegated to any other object by virtual keyword means.

Q: What is constructor or ctor?
A: Constructor creates an object and initializes it. It also creates vtable for virtual functions. It is different from other methods in a class.

Q: What about Virtual Destructor?
A: Yes there is a Virtual Destructor. A destructor can be virtual as it is possible as at runtime depending on the type of object caller is calling to, proper destructor will be called.

Q: What is the difference between a copy constructor and an overloaded assignment operator?
A: 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.
Q: Can a constructor throws an exception? How to handle the error when the constructor fails?
A:The constructor never throws an error.

Q: What is default constructor?
A: Constructor with no arguments or all the arguments has default values.

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

Q: When are copy constructors called?
A: Copy constructors are called in following cases:
(a) when a function returns an object of that class by value
(b) when the object of that class is passed by value as an argument to a function
(c) when you construct an object based on another object of the same class
(d) When compiler generates a temporary object

Q: Can a copy constructor accept an object of the same class as parameter, instead of reference of the object?
A: No. It is specified in the definition of the copy constructor itself. It should generate an error if a programmer specifies a copy constructor with a first argument that is an object and not a reference.

Q: What is conversion constructor?
A: constructor with a single argument makes that constructor as conversion ctor and it can be used for type conversion.
for example:

```
class Boo
{
public:
Boo( int i );
};
Boo BooObject = 10 ; // assigning int 10 Boo object
```

Q:What is conversion operator??
A:class can have a public method for specific data type conversions.
for example:

```
class Boo
{
double value;
public:
Boo(int i )
operator double()
{
return value;
}
};
Boo BooObject;
double i = BooObject; // assigning object to variable i of type double.
```

now conversion operator gets called to assign the value.

Q: How can I handle a constructor that fails?
A: throw an exception. Constructors don't have a return type, so it's not possible to use return codes. The best way to signal constructor failure is therefore to throw an exception.

Q: How can I handle a destructor that fails?
A: Write a message to a log-_le. But do not throw an exception. The C++ rule is that you must never throw an exception from a destructor that is being called during the "stack unwinding" process of another exception. For example, if someone says throw Foo(), the stack will be unwound so all the stack frames between the throw Foo() and the } catch (Foo e) { will get popped. This is called stack unwinding. During stack unwinding, all the local objects in all

those stack frames are destructed. If one of those destructors throws an exception (say it throws a Bar object), the C++ runtime system is in a no-win situation: should it ignore the Bar and end up in the } catch (Foo e) { where it was originally headed? Should it ignore the Foo and look for a } catch (Bare) { handler? There is no good answer:either choice
loses information. So the C++ language guarantees that it will call terminate() at this point, and terminate() kills the process. Bang you're dead.

Q: What is Virtual Destructor?
A: Using virtual destructors, you can destroy objects without knowing their type - the correct destructor for the object is invoked using the virtual function mechanism. Note that destructors can also be declared as pure virtual functions for abstract classes. if someone will derive from your class, and if someone will say "new Derived", where "Derived" is derived from your class, and if someone will say delete p, where the actual object's type is "Derived" but the pointer p's type is your class.

Q: Can a copy constructor accept an object of the same class as parameter, instead of reference of the object?
A: No. It is specified in the definition of the copy constructor itself. It should generate an error if a programmer specifies a copy constructor with a first argument that is an object and not a reference.

Q: What's the order that local objects are destructed?
A: In reverse order of construction: First constructed, last destructed.

In the following example, b's destructor will be executed first, then a's destructor:

```
void userCode()
{
Fred a;
Fred b;
...
}
```

Q: What's the order that objects in an array are destructed?
A: In reverse order of construction: First constructed, last destructed.

In the following example, the order for destructors will be a[9], a[8], ..., a[1], a[0]:

```
void userCode()
{
Fred a[10];
...
}
```

Q: Can I overload the destructor for my class?

A: No.

You can have only one destructor for a class Fred. It's always called Fred::~Fred(). It never takes any parameters, and it never returns anything.

You can't pass parameters to the destructor anyway, since you never explicitly call a destructor (well, almost never).

Q: Should I explicitly call a destructor on a local variable?
A: No!

The destructor will get called again at the close } of the block in which the local was created. This is a guarantee of the language; it happens automagically; there's no way to stop it from happening. But you can get really bad results from calling a destructor on the same object a second time! Bang! You're dead!

Q: 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?
A: No! [For context, please read the previous FAQ].

Suppose the (desirable) side effect of destructing a local File object is to close the File. Now suppose you have an object f of a class File and you want File f to be closed before the end of the scope (i.e., the }) of the scope of object f:

```
void someCode()
{
File f;

...insert code that should execute when f is still open...

We want the side-effect of f's destructor here!

...insert code that should execute after f is closed...
}
```

There is a simple solution to this problem. But in the mean time, remember: Do not explicitly call the destructor!

Q: OK, OK already; I won't explicitly call the destructor of a local; but how do I handle the above situation?
A: Simply wrap the extent of the lifetime of the local in an artificial block {...}:

```
void someCode()
{
{
```

```
File f;
...insert code that should execute when f is still open...
} f's destructor will automagically be called here!

...insert code here that should execute after f is closed...}
```

Q: What if I can't wrap the local in an artificial block?
A: Most of the time, you can limit the lifetime of a local by wrapping the local in an artificial block ({...}). But if for some reason you can't do that, add a member function that has a similar effect as the destructor. But do not call the destructor itself!

For example, in the case of class File, you might add a close() method. Typically the destructor will simply call this close() method. Note that the close() method will need to mark the File object so a subsequent call won't re-close an already-closed File. E.g., it might set the fileHandle_ data member to some nonsensical value such as -1, and it might check at the beginning to see if the fileHandle_ is already equal to -1:

```
class File {
public:
void close();
~File();
...
private:
int fileHandle_; // fileHandle_ >= 0 if/only-if it's open
};

File::~File()
{
close();
}

void File::close()
{
if (fileHandle_ >= 0) {
...insert code to call the OS to close the file...
fileHandle_ = -1;
}
}
```

Note that the other File methods may also need to check if the fileHandle_ is -1 (i.e., check if the File is closed).

Note also that any constructors that don't actually open a file should set fileHandle_ to -1.

Q: But can I explicitly call a destructor if I've allocated my object with new?
A: Probably not.

Unless you used placement new, you should simply delete the object rather than explicitly calling the destructor. For example, suppose you allocated the object via a typical new expression:

```
Fred* p = new Fred();
```

Then the destructor Fred::~Fred() will automagically get called when you delete it via:

```
delete p; // Automagically calls p->~Fred()
```

You should not explicitly call the destructor, since doing so won't release the memory that was allocated for the Fred object itself. Remember: delete p does two things: it calls the destructor and it deallocates the memory.

Q: What is "placement new" and why would I use it?
A: There are many uses of placement new. The simplest use is to place an object at a particular location in memory. This is done by supplying the place as a pointer parameter to the new part of a new expression:

```
#include // Must #include this to use "placement new"
#include "Fred.h" // Declaration of class Fred

void someCode()
{
char memory[sizeof(Fred)]; // Line #1
void* place = memory; // Line #2

Fred* f = new(place) Fred(); // Line #3 (see "DANGER" below)
// The pointers f and place will be equal

...
}
```

Line #1 creates an array of sizeof(Fred) bytes of memory, which is big enough to hold a Fred object. Line #2 creates a pointer place that points to the first byte of this memory (experienced C programmers will note that this step was unnecessary; it's there only to make the code more obvious). Line #3 essentially just calls the constructor Fred::Fred(). The this pointer in the Fred constructor will be equal to place. The returned pointer f will therefore be equal to place.

ADVICE: Don't use this "placement new" syntax unless you have to. Use it only when you really care that an object is placed at a particular location in memory. For example, when your hardware has a memory-mapped I/O timer device, and you want to place a Clock object at that memory location.

DANGER: You are taking sole responsibility that the pointer you pass to the "placement new"

operator points to a region of memory that is big enough and is properly aligned for the object type that you're creating. Neither the compiler nor the run-time system make any attempt to check whether you did this right. If your Fred class needs to be aligned on a 4 byte boundary but you supplied a location that isn't properly aligned, you can have a serious disaster on your hands (if you don't know what "alignment" means, please don't use the placement new syntax). You have been warned.

You are also solely responsible for destructing the placed object. This is done by explicitly calling the destructor:

```
void someCode()
{
char memory[sizeof(Fred)];
void* p = memory;
Fred* f = new(p) Fred();
...
f->~Fred(); // Explicitly call the destructor for the placed object
}
```

This is about the only time you ever explicitly call a destructor.

Note: there is a much cleaner but more sophisticated way of handling the destruction / deletion situation.

Q: When I write a destructor, do I need to explicitly call the destructors for my member objects?
A: No. You never need to explicitly call a destructor (except with placement new).

A class's destructor (whether or not you explicitly define one) automagically invokes the destructors for member objects. They are destroyed in the reverse order they appear within the declaration for the class.

```
class Member {
public:
~Member();
...
};

class Fred {
public:
~Fred();
...
private:
Member x_;
Member y_;
Member z_;
```

```
};

Fred::~Fred()
{
// Compiler automagically calls z_.~Member()
// Compiler automagically calls y_.~Member()
// Compiler automagically calls x_.~Member()
}
```

Q: When I write a derived class's destructor, do I need to explicitly call the destructor for my base class?
A: No. You never need to explicitly call a destructor (except with placement new).

A derived class's destructor (whether or not you explicitly define one) automagically invokes the destructors for base class subobjects. Base classes are destructed after member objects. In the event of multiple inheritance, direct base classes are destructed in the reverse order of their appearance in the inheritance list.

```
class Member {
public:
~Member();
...
};

class Base {
public:
virtual ~Base(); // A virtual destructor
...
};

class Derived : public Base {
public:
~Derived();
...
private:
Member x_;
};

Derived::~Derived()
{
// Compiler automagically calls x_.~Member()
// Compiler automagically calls Base::~Base()
}
```

Note: Order dependencies with virtual inheritance are trickier. If you are relying on order dependencies in a virtual inheritance hierarchy, you'll need a lot more information than is in this

FAQ.

Q: Is there any difference between List x; and List x();?
A: A big difference!

Suppose that List is the name of some class. Then function f() declares a local List object called x:

```
void f()
{
List x; // Local object named x (of class List)
...
}
```

But function g() declares a function called x() that returns a List:

```
void g()
{
List x(); // Function named x (that returns a List)
...
}
```

Q: Can one constructor of a class call another constructor of the same class to initialize the this object?
A: Nope.

Let's work an example. Suppose you want your constructor Foo::Foo(char) to call another constructor of the same class, say Foo::Foo(char,int), in order that Foo::Foo(char,int) would help initialize the this object. Unfortunately there's no way to do this in C++.

Some people do it anyway. Unfortunately it doesn't do what they want. For example, the line Foo(x, 0); does not call Foo::Foo(char,int) on the this object. Instead it calls Foo::Foo(char,int) to initialize a temporary, local object (not this), then it immediately destructs that temporary when control flows over the ;.

```
class Foo {
public:
Foo(char x);
Foo(char x, int y);
...
};

Foo::Foo(char x)
{
```

```
...
Foo(x, 0); // this line does NOT help initialize the this object!!
...
}
```
You can sometimes combine two constructors via a default parameter:

```
class Foo {
public:
Foo(char x, int y=0); // this line combines the two constructors
...
};
```
If that doesn't work, e.g., if there isn't an appropriate default parameter that combines the two constructors, sometimes you can share their common code in a private init() member function:

```
class Foo {
public:
Foo(char x);
Foo(char x, int y);
...
private:
void init(char x, int y);
};

Foo::Foo(char x)
{
init(x, int(x) + 7);
...
}

Foo::Foo(char x, int y)
{
init(x, y);
...
}

void Foo::init(char x, int y)
{
...
}
```
BTW do NOT try to achieve this via placement new. Some people think they can say new(this) Foo(x, int(x)+7) within the body of Foo::Foo(char). However that is bad, bad, bad. Please don't write me and tell me that it seems to work on your particular version of your particular compiler; it's bad. Constructors do a bunch of little magical things behind the scenes, but that bad technique steps on those partially constructed bits. Just say no.

Q: Is the default constructor for Fred always Fred::Fred()?
A: No. A "default constructor" is a constructor that can be called with no arguments. One example of this is a constructor that takes no parameters:

```
class Fred {
public:
Fred(); // Default constructor: can be called with no args
...
};
```

Another example of a "default constructor" is one that can take arguments, provided they are given default values:

```
class Fred {
public:
Fred(int i=3, int j=5); // Default constructor: can be called with no args
...
};
```

Q: Which constructor gets called when I create an array of Fred objects?
A: Fred's default constructor (except as discussed below).

```
class Fred {
public:
Fred();
...
};

int main()
{
Fred a[10];  calls the default constructor 10 times
Fred* p = new Fred[10];  calls the default constructor 10 times
...
}
```

If your class doesn't have a default constructor, you'll get a compile-time error when you attempt to create an array using the above simple syntax:

```
class Fred {
public:
Fred(int i, int j); assume there is no default constructor
...
};
```

```
int main()
{
Fred a[10]; ERROR: Fred doesn't have a default constructor
Fred* p = new Fred[10];  ERROR: Fred doesn't have a default constructor
...
}
```

However, even if your class already has a default constructor, you should try to use std::vector rather than an array (arrays are evil). std::vector lets you decide to use any constructor, not just the default constructor:

```
#include

int main()
{
std::vector a(10, Fred(5,7));  the 10 Fred objects in std::vector a will be initialized with Fred(5,7)
...
}
```

Even though you ought to use a std::vector rather than an array, there are times when an array might be the right thing to do, and for those, you might need the "explicit initialization of arrays" syntax. Here's how:

```
class Fred {
public:
Fred(int i, int j);  assume there is no default constructor
...
};

int main()
{
Fred a[10] = {
Fred(5,7), Fred(5,7), Fred(5,7), Fred(5,7), Fred(5,7), // The 10 Fred objects are
Fred(5,7), Fred(5,7), Fred(5,7), Fred(5,7), Fred(5,7) // initialized using Fred(5,7)
};
...
}
```

Of course you don't have to do Fred(5,7) for every entry you can put in any numbers you want, even parameters or other variables.

Finally, you can use placement-new to manually initialize the elements of the array. Warning: it's ugly: the raw array can't be of type Fred, so you'll need a bunch of pointer-casts to do things like compute array index operations. Warning: it's compiler- and hardware-dependent: you'll need to make sure the storage is aligned with an alignment that is at least as strict as is required for objects of class Fred. Warning: it's tedious to make it exception-safe: you'll need to manually

destruct the elements, including in the case when an exception is thrown part-way through the loop that calls the constructors. But if you really want to do it anyway, read up on placement-new. (BTW placement-new is the magic that is used inside of std::vector. The complexity of getting everything right is yet another reason to use std::vector.)

By the way, did I ever mention that arrays are evil? Or did I mention that you ought to use a std::vector unless there is a compelling reason to use an array?

Q: Should my constructors use "initialization lists" or "assignment"?
A: Initialization lists. In fact, constructors should initialize as a rule all member objects in the initialization list. One exception is discussed further down.

Consider the following constructor that initializes member object x_ using an initialization list: Fred::Fred() : x_(whatever) { }. The most common benefit of doing this is improved performance. For example, if the expression whatever is the same type as member variable x_, the result of the whatever expression is constructed directly inside x_ the compiler does not make a separate copy of the object. Even if the types are not the same, the compiler is usually able to do a better job with initialization lists than with assignments.

The other (inefficient) way to build constructors is via assignment, such as: Fred::Fred() { x_ = whatever; }. In this case the expression whatever causes a separate, temporary object to be created, and this temporary object is passed into the x_ object's assignment operator. Then that temporary object is destructed at the ;. That's inefficient.

As if that wasn't bad enough, there's another source of inefficiency when using assignment in a constructor: the member object will get fully constructed by its default constructor, and this might, for example, allocate some default amount of memory or open some default file. All this work could be for naught if the whatever expression and/or assignment operator causes the object to close that file and/or release that memory (e.g., if the default constructor didn't allocate a large enough pool of memory or if it opened the wrong file).

Conclusion: All other things being equal, your code will run faster if you use initialization lists rather than assignment.

Note: There is no performance difference if the type of x_ is some built-in/intrinsic type, such as int or char* or float. But even in these cases, my personal preference is to set those data members in the initialization list rather than via assignment for consistency. Another symmetry argument in favor of using initialization lists even for built-in/intrinsic types: non-static const and non-static reference data members can't be assigned a value in the constructor, so for symmetry it makes sense to initialize everything in the initialization list.

Now for the exceptions. Every rule has exceptions (hmmm; does "every rule has exceptions" have exceptions? reminds me of Gdel's Incompleteness Theorems), and there are a couple of exceptions to the "use initialization lists" rule. Bottom line is to use common sense: if it's cheaper, better, faster, etc. to not use them, then by all means, don't use them. This might happen when your class has two constructors that need to initialize the this object's data members in

different orders. Or it might happen when two data members are self-referential. Or when a data-member needs a reference to the this object, and you want to avoid a compiler warning about using the this keyword prior to the { that begins the constructor's body (when your particular compiler happens to issue that particular warning). Or when you need to do an if/throw test on a variable (parameter, global, etc.) prior to using that variable to initialize one of your this members. This list is not exhaustive; please don't write me asking me to add another "Or when...". The point is simply this: use common sense.

Q: Should you use the this pointer in the constructor?
A: Some people feel you should not use the this pointer in a constructor because the object is not fully formed yet. However you can use this in the constructor (in the {body} and even in the initialization list) if you are careful.

Here is something that always works: the {body} of a constructor (or a function called from the constructor) can reliably access the data members declared in a base class and/or the data members declared in the constructor's own class. This is because all those data members are guaranteed to have been fully constructed by the time the constructor's {body} starts executing.

Here is something that never works: the {body} of a constructor (or a function called from the constructor) cannot get down to a derived class by calling a virtual member function that is overridden in the derived class. If your goal was to get to the overridden function in the derived class, you won't get what you want. Note that you won't get to the override in the derived class independent of how you call the virtual member function: explicitly using the this pointer (e.g., this->method()), implicitly using the this pointer (e.g., method()), or even calling some other function that calls the virtual member function on your this object. The bottom line is this: even if the caller is constructing an object of a derived class, during the constructor of the base class, your object is not yet of that derived class. You have been warned.

Here is something that sometimes works: if you pass any of the data members in this object to another data member's initializer, you must make sure that the other data member has already been initialized. The good news is that you can determine whether the other data member has (or has not) been initialized using some straightforward language rules that are independent of the particular compiler you're using. The bad news it that you have to know those language rules (e.g., base class sub-objects are initialized first (look up the order if you have multiple and/or virtual inheritance!), then data members defined in the class are initialized in the order in which they appear in the class declaration). If you don't know these rules, then don't pass any data member from the this object (regardless of whether or not you explicitly use the this keyword) to any other data member's initializer! And if you do know the rules, please be careful.

Q: What is the "Named Constructor Idiom"?
A: A technique that provides more intuitive and/or safer construction operations for users of your class.

The problem is that constructors always have the same name as the class. Therefore the only way to differentiate between the various constructors of a class is by the parameter list. But if there are lots of constructors, the differences between them become somewhat subtle and error prone.

With the Named Constructor Idiom, you declare all the class's constructors in the private or protected sections, and you provide public static methods that return an object. These static methods are the so-called "Named Constructors." In general there is one such static method for each different way to construct an object.

For example, suppose we are building a Point class that represents a position on the X-Y plane. Turns out there are two common ways to specify a 2-space coordinate: rectangular coordinates (X+Y), polar coordinates (Radius+Angle). (Don't worry if you can't remember these; the point isn't the particulars of coordinate systems; the point is that there are several ways to create a Point object.) Unfortunately the parameters for these two coordinate systems are the same: two floats. This would create an ambiguity error in the overloaded constructors:

```
class Point {
public:
Point(float x, float y); // Rectangular coordinates
Point(float r, float a); // Polar coordinates (radius and angle)
// ERROR: Overload is Ambiguous: Point::Point(float,float)
};

int main()
{
Point p = Point(5.7, 1.2); // Ambiguous: Which coordinate system?
...
}
```

One way to solve this ambiguity is to use the Named Constructor Idiom:

```
#include // To get sin() and cos()

class Point {
public:
static Point rectangular(float x, float y); // Rectangular coord's
static Point polar(float radius, float angle); // Polar coordinates
// These static methods are the so-called "named constructors"
...
private:
Point(float x, float y); // Rectangular coordinates
float x_, y_;
};

inline Point::Point(float x, float y)
: x_(x), y_(y) { }

inline Point Point::rectangular(float x, float y)
```

```
{ return Point(x, y); }

inline Point Point::polar(float radius, float angle)
{ return Point(radius*cos(angle), radius*sin(angle)); }
```
Now the users of Point have a clear and unambiguous syntax for creating Points in either coordinate system:

```
int main()
{
Point p1 = Point::rectangular(5.7, 1.2); // Obviously rectangular
Point p2 = Point::polar(5.7, 1.2); // Obviously polar
...
}
```
Make sure your constructors are in the protected section if you expect Point to have derived classes.

The Named Constructor Idiom can also be used to make sure your objects are always created via new.

Note that the Named Constructor Idiom, at least as implemented above, is just as fast as directly calling a constructor modern compilers will not make any extra copies of your object.

Virtual

Q: What is virtual function?
A: When derived class overrides the base class method by redefining the same function, then if client wants to access redefined the method from derived class through a pointer from base class object, then you must define this function in base class as virtual function.

```
class parent
{
void Show()
{
cout << "i'm parent" << endl;
}
};
```

```
class child: public parent
{
void Show()
{
cout << "i'm child" << endl;
}
};
parent * parent_object_ptr = new child;
parent_object_ptr->show() // calls parent->show()
```

now we goto virtual world...

```
class parent
{
virtual void Show()
{
cout << "i'm parent" << endl;
}
};
class child: public parent
{
void Show()
{
cout << "i'm child" << endl;
}
};
parent * parent_object_ptr = new child;
parent_object_ptr->show() // calls child->show()
```

Q: What is a "pure virtual" member function?
A: The abstract class whose pure virtual method has to be implemented by all the classes which derive on these. Otherwise it would result in a compilation error. This construct should be used when one wants to ensure that all the derived classes implement the method defined as pure virtual in base class.

Q: How virtual functions are implemented C++?
A: Virtual functions are implemented using a table of function pointers, called the vtable. There is one entry in the table per virtual function in the class. This table is created by the constructor of the class. When a derived class is constructed, its base class is constructed _rst which creates the vtable. If the derived class overrides any of the base classes virtual functions, those entries in the vtable are overwritten by the derived class constructor. This is why you should never call virtual functions from a constructor: because the vtable entries for the object may not have been set up by the derived class constructor yet, so you might end up calling base class implementations of those virtual functions

Q: What is pure virtual function? or what is abstract class?
A: When you de_ne only function prototype in a base class without implementation and do the complete implementation in derived class. This base class is called abstract class and client won't able to instantiate an object using this base class. You can make a pure virtual function or abstract class this way..

```
class Boo
{
void foo() = 0;
}
Boo MyBoo; // compilation error
```

Q: What is Pure Virtual Function? Why and when it is used?
A: The abstract class whose pure virtual method has to be implemented by all the classes which derive on these. Otherwise it would result in a compilation error. This construct should be used when one wants to ensure that all the derived classes implement the method defined as pure virtual in base class.

Q: How Virtual functions call up is maintained?
A: Through Look up tables added by the compile to every class image. This also leads to performance penalty.

Q: What is a virtual destructor?
A: The simple answer is that a virtual destructor is one that is declared with the virtual attribute. The behavior of a virtual destructor is what is important. If you destroy an object through a caller or reference to a base class, and the base-class destructor is not virtual, the derived-class destructors are not executed, and the destruction might not be complete.

Inheritance

Q: What is inheritance?
A: Inheritance allows one class to reuse the state and behavior of another class. The derived class inherits the properties and method implementations of the base class and extends it by overriding methods and adding additional properties and methods.

Q: When should you use multiple inheritance?
A:There are three acceptable answers:- "Never,""Rarely," and "When the problem domain cannot be accurately modeled any other way." Consider an Asset class, Building class, Vehicle class, and CompanyCar class. All company cars are vehicles. Some company cars are assets because the organizations own them. Others might be leased. Not all assets are vehicles. Money accounts are assets. Real estate holdings are assets. Some real estate holdings are buildings. Not all buildings are assets. Ad infinitum. When you diagram these relationships, it becomes apparent that multiple inheritance is a likely and intuitive way to model this common problem domain. The applicant should understand, however, that multiple inheritance, like a chainsaw, is a useful tool that has its perils, needs respect, and is best avoided except when nothing else will do.

Q: Explain the ISA and HASA class relationships. How would you implement each in a class design?
A: A specialized class "is" a specialization of another class and, therefore, has the ISA relationship with the other class. This relationship is best implemented by embedding an object of the Salary class in the Employee class.

Q: When is a template a better solution than a base class?
A: 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 generality) to the designer of the container or manager class.

Q: What is multiple inheritance(virtual inheritance)? What are its advantages and disadvantages?
A: Multiple Inheritance is the process whereby a child can be derived from more than one parent class. The advantage of multiple inheritance is that it allows a class to inherit the functionality of more than one base class thus allowing for modeling of complex relationships.
The disadvantage of multiple inheritance is that it can lead to a lot of confusion(ambiguity) when two base classes implement a method with the same name.

Q: What a derived class inherits or doesn't inherit?
A: Inherits:

Every data member defined in the parent class (although such members may not always be accessible in the derived class!)

Every ordinary member function of the parent class (although such members may not always be accessible in the derived class!)

The same initial data layout as the base class.

Doesn't Inherit :

The base class's constructors and destructor.

The base class's assignment operator.

The base class's friends

Polymorphism

Q: What is Polymorphism??
A: Polymorphism allows a client to treat di_erent objects in the same way even if they were created from di_erent classes and exhibit di_erent behaviors. You can use implementation inheritance to achieve polymorphism in languages such as C++ and Java. Base class object's pointer can invoke methods in derived class objects. You can also achieve polymorphism in C++ by function overloading and operator overloading.

Q: What is problem with Runtime type identification?
A: The run time type identification comes at a cost of performance penalty. Compiler maintains the class.

Q: What is Polymorphism?
A: Polymorphism allows a client to treat different objects in the same way even if they were created from different classes and exhibit different behaviors. You can use implementation inheritance to achieve polymorphism in languages such as C++ and Java. Base class object's pointer can invoke methods in derived class objects. You can also achieve polymorphism in C++ by function overloading and operator overloading.

Classes

Q: What is a class?
A: A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

Q: What are the differences between a C++ struct and C++ class?
A: The default member and base class access specifies are different. This is one of the commonly misunderstood aspects of C++. Believe it or not, many programmers think that a C++ struct is just like a C struct, while a C++ class has inheritance, access specifes, member functions, overloaded operators, and so on. Actually, the C++ struct has all the features of the class. The only differences are that a struct defaults to public member access and public base class inheritance, and a class defaults to the private access specified and private base-class inheritance.

Q: How do you know that your class needs a virtual destructor?
A: If your class has at least one virtual function, you should make a destructor for this class virtual. This will allow you to delete a dynamic object through a caller to a base class object. If the destructor is non-virtual, then wrong destructor will be invoked during deletion of the dynamic object.

Q: What is encapsulation?
A: Containing and hiding Information about an object, such as internal data structures and code. Encapsulation isolates the internal complexity of an object's operation from the rest of the application. For example, a client component asking for net revenue from a business object need not know the data's origin.

Q: What is "this" pointer?
A: The this pointer is a pointer accessible only within the member functions of a class, struct, or union type. It points to the object for which the member function is called. Static member functions do not have a this pointer. When a nonstatic member function is called for an object, the address of the object is passed as a hidden argument to the function. For example, the following function call
myDate.setMonth(3);
can be interpreted this way:
setMonth(&myDate, 3);
The object's address is available from within the member function as the this pointer. It is legal, though unnecessary, to use the this pointer when referring to members of the class.

Q: What happens when you make call "delete this;"?
A: The code has two built-in pitfalls. First, if it executes in a member function for an extern, static, or automatic object, the program will probably crash as soon as the delete statement executes. There is no portable way for an object to tell that it was instantiated on the heap, so the class cannot assert that its object is properly instantiated. Second, when an object commits suicide this way, the using program might not know about its demise. As far as the instantiating program is concerned, the object remains in scope and continues to exist even though the object

did itself in. Subsequent dereferencing of the pointer can and usually does lead to disaster.
You should never do this. Since compiler does not know whether the object was allocated on the stack or on the heap, "delete this" could cause a disaster.

Q: What is assignment operator?
A: Default assignment operator handles assigning one object to another of the same class.
Member to member copy (shallow copy)

Q: What are all the implicit member functions of the class? Or what are all the functions which compiler implements for us if we don't define one?
A:
(a) default ctor
(b) copy ctor
(c) assignment operator
(d) default destructor
(e) address operator

Q: What is a container class? What are the types of container classes?
A: 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.

Q: What is Overriding?
A: To override a method, a subclass of the class that originally declared the method must declare a method with the same name, return type (or a subclass of that return type), and same parameter list.
The definition of the method overriding is:
· Must have same method name.
· Must have same data type.
· Must have same argument list.
Overriding a method means that replacing a method functionality in child class. To imply overriding functionality we need parent and child classes. In the child class you define the same method signature as one defined in the parent class.

Q: How do you access the static member of a class?
A: ::

Q: What is a nested class? Why can it be useful?
A: A nested class is a class enclosed within the scope of another class. For example:
```
// Example 1: Nested class
//
class OuterClass
```

```
{
class NestedClass
{
// ...
};
// ...
};
```
Nested classes are useful for organizing code and controlling access and dependencies. Nested classes obey access rules just like other parts of a class do; so, in Example 1, if NestedClass is public then any code can name it as OuterClass::NestedClass. Often nested classes contain private implementation details, and are therefore made private; in Example 1, if NestedClass is private, then only OuterClass's members and friends can use NestedClass. When you instantiate as outer class, it won't instantiate inside class.

Q: What is a local class? Why can it be useful?
A: Local class is a class defined within the scope of a function _ any function, whether a member function or a free function. For example:

```
// Example 2: Local class
//
int f()
{
class LocalClass
{
// ...
};
// ...
};
```

Like nested classes, local classes can be a useful tool for managing code dependencies.

Q: What a derived class can add?
A: New data members

New member functions

New constructors and destructor

New friends

Q: What happens when a derived-class object is created and destroyed?
A: Space is allocated (on the stack or the heap) for the full object (that is, enough space to store the data members inherited from the base class plus the data members defined in the derived class itself)

The base class's constructor is called to initialize the data members inherited from the base class

The derived class's constructor is then called to initialize the data members added in the derived class

The derived-class object is then usable

When the object is destroyed (goes out of scope or is deleted) the derived class's destructor is called on the object first

Then the base class's destructor is called on the object

Finally the allocated space for the full object is reclaimed

Q: How do I create a subscript operator for a Matrix class?
A: Use operator() rather than operator[].

When you have multiple subscripts, the cleanest way to do it is with operator() rather than with operator[]. The reason is that operator[] always takes exactly one parameter, but operator() can take any number of parameters (in the case of a rectangular matrix, two parameters are needed).

For example:

```
class Matrix {
public:
Matrix(unsigned rows, unsigned cols);
double& operator() (unsigned row, unsigned col);  subscript operators often come in pairs
double operator() (unsigned row, unsigned col) const;  subscript operators often come in pairs
...
~Matrix(); // Destructor
Matrix(const Matrix& m); // Copy constructor
Matrix& operator= (const Matrix& m); // Assignment operator
...
private:
unsigned rows_, cols_;
double* data_;
};

inline
Matrix::Matrix(unsigned rows, unsigned cols)
: rows_ (rows)
, cols_ (cols)
//data_ <--initialized below (after the 'if/throw' statement)
{
if (rows == 0 || cols == 0)
throw BadIndex("Matrix constructor has 0 size");
data_ = new double[rows * cols];
}
```

```
inline
Matrix::~Matrix()
{
delete[] data_;
}

inline
double& Matrix::operator() (unsigned row, unsigned col)
{
if (row >= rows_ || col >= cols_)
throw BadIndex("Matrix subscript out of bounds");
return data_[cols_*row + col];
}

inline
double Matrix::operator() (unsigned row, unsigned col) const
{
if (row >= rows_ || col >= cols_)
throw BadIndex("const Matrix subscript out of bounds");
return data_[cols_*row + col];
}
```
Then you can access an element of Matrix m using m(i,j) rather than m[i][j]:

```
int main()
{
Matrix m(10,10);
m(5,8) = 106.15;
std::cout << m(5,8);
...
}
```

Q: Why shouldn't my Matrix class's interface look like an array-of-array?
A: Here's what this FAQ is really all about: Some people build a Matrix class that has an operator[] that returns a reference to an Array object (or perhaps to a raw array, shudder), and that Array object has an operator[] that returns an element of the Matrix (e.g., a reference to a double). Thus they access elements of the matrix using syntax like m[i][j] rather than syntax like m(i,j).

The array-of-array solution obviously works, but it is less flexible than the operator() approach. Specifically, there are easy performance tuning tricks that can be done with the operator() approach that are more difficult in the [][] approach, and therefore the [][] approach is more likely to lead to bad performance, at least in some cases.

For example, the easiest way to implement the [][] approach is to use a physical layout of the matrix as a dense matrix that is stored in row-major form (or is it column-major; I can't ever

remember). In contrast, the operator() approach totally hides the physical layout of the matrix, and that can lead to better performance in some cases.

Put it this way: the operator() approach is never worse than, and sometimes better than, the [][] approach.

The operator() approach is never worse because it is easy to implement the dense, row-major physical layout using the operator() approach, so when that configuration happens to be the optimal layout from a performance standpoint, the operator() approach is just as easy as the [][] approach (perhaps the operator() approach is a tiny bit easier, but I won't quibble over minor nits).
The operator() approach is sometimes better because whenever the optimal layout for a given application happens to be something other than dense, row-major, the implementation is often significantly easier using the operator() approach compared to the [][] approach.
As an example of when a physical layout makes a significant difference, a recent project happened to access the matrix elements in columns (that is, the algorithm accesses all the elements in one column, then the elements in another, etc.), and if the physical layout is row-major, the accesses can "stride the cache". For example, if the rows happen to be almost as big as the processor's cache size, the machine can end up with a "cache miss" for almost every element access. In this particular project, we got a 20% improvement in performance by changing the mapping from the logical layout (row,column) to the physical layout (column,row).

Of course there are many examples of this sort of thing from numerical methods, and sparse matrices are a whole other dimension on this issue. Since it is, in general, easier to implement a sparse matrix or swap row/column ordering using the operator() approach, the operator() approach loses nothing and may gain something it has no down-side and a potential up-side.

Use the operator() approach.

Q: Should I design my classes from the outside (interfaces first) or from the inside (data first)?
A: From the outside!

A good interface provides a simplified view that is expressed in the vocabulary of a user. In the case of OO software, the interface is normally the set of public methods of either a single class or a tight group of classes.

First think about what the object logically represents, not how you intend to physically build it. For example, suppose you have a Stack class that will be built by containing a LinkedList:

```
class Stack {
public:
...
private:
LinkedList list_;
};
```

Should the Stack have a get() method that returns the LinkedList? Or a set() method that takes a LinkedList? Or a constructor that takes a LinkedList? Obviously the answer is No, since you should design your interfaces from the outside-in. I.e., users of Stack objects don't care about LinkedLists; they care about pushing and popping.

Now for another example that is a bit more subtle. Suppose class LinkedList is built using a linked list of Node objects, where each Node object has a pointer to the next Node:

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

class LinkedList {
public:
...
private:
Node* first_;
};
```

Should the LinkedList class have a get() method that will let users access the first Node? Should the Node object have a get() method that will let users follow that Node to the next Node in the chain? In other words, what should a LinkedList look like from the outside? Is a LinkedList really a chain of Node objects? Or is that just an implementation detail? And if it is just an implementation detail, how will the LinkedList let users access each of the elements in the LinkedList one at a time?

The key insight is the realization that a LinkedList is not a chain of Nodes. That may be how it is built, but that is not what it is. What it is is a sequence of elements. Therefore the LinkedList abstraction should provide a LinkedListIterator class as well, and that LinkedListIterator might have an operator++ to go to the next element, and it might have a get()/set() pair to access its value stored in the Node (the value in the Node element is solely the responsibility of the LinkedList user, which is why there is a get()/set() pair that allows the user to freely manipulate that value).

Starting from the user's perspective, we might want our LinkedList class to support operations that look similar to accessing an array using pointer arithmetic:

```
void userCode(LinkedList& a)
{
for (LinkedListIterator p = a.begin(); p != a.end(); ++p)
std::cout << *p << '\n';
}
```

To implement this interface, LinkedList will need a begin() method and an end() method. These return a LinkedListIterator object. The LinkedListIterator will need a method to go forward, ++p; a method to access the current element, *p; and a comparison operator, p != a.end().

The code follows. The important thing to notice is that LinkedList does not have any methods

that let users access Nodes. Nodes are an implementation technique that is completely buried. This makes the LinkedList class safer (no chance a user will mess up the invariants and linkages between the various nodes), easier to use (users don't need to expend extra effort keeping the node-count equal to the actual number of nodes, or any other infrastructure stuff), and more flexible (by changing a single typedef, users could change their code from using LinkedList to some other list-like class and the bulk of their code would compile cleanly and hopefully with improved performance characteristics).

```
#include // Poor man's exception handling

class LinkedListIterator;
class LinkedList;

class Node {
// No public members; this is a "private class"
friend class LinkedListIterator; // A friend class
friend class LinkedList;
Node* next_;
int elem_;
};

class LinkedListIterator {
public:
bool operator== (LinkedListIterator i) const;
bool operator!= (LinkedListIterator i) const;
void operator++ (); // Go to the next element
int& operator* (); // Access the current element
private:
LinkedListIterator(Node* p);
Node* p_;
friend class LinkedList; // so LinkedList can construct a LinkedListIterator
};

class LinkedList {
public:
void append(int elem); // Adds elem after the end
void prepend(int elem); // Adds elem before the beginning
...
LinkedListIterator begin();
LinkedListIterator end();
...
private:
Node* first_;
};
```

Here are the methods that are obviously inlinable (probably in the same header file):

```
inline bool LinkedListIterator::operator== (LinkedListIterator i) const
{
return p_ == i.p_;
}

inline bool LinkedListIterator::operator!= (LinkedListIterator i) const
{
return p_ != i.p_;
}

inline void LinkedListIterator::operator++()
{
assert(p_ != NULL); // or if (p_==NULL) throw ...
p_ = p_->next_;
}

inline int& LinkedListIterator::operator*()
{
assert(p_ != NULL); // or if (p_==NULL) throw ...
return p_->elem_;
}

inline LinkedListIterator::LinkedListIterator(Node* p)
: p_(p)
{ }

inline LinkedListIterator LinkedList::begin()
{
return first_;
}

inline LinkedListIterator LinkedList::end()
{
return NULL;
}
```

Conclusion: The linked list had two different kinds of data. The values of the elements stored in the linked list are the responsibility of the user of the linked list (and only the user; the linked list itself makes no attempt to prohibit users from changing the third element to 5), and the linked list's infrastructure data (next pointers, etc.), whose values are the responsibility of the linked list (and only the linked list; e.g., the linked list does not let users change (or even look at!) the various next pointers).

Thus the only get()/set() methods were to get and set the elements of the linked list, but not the infrastructure of the linked list. Since the linked list hides the infrastructure pointers/etc., it is

able to make very strong promises regarding that infrastructure (e.g., if it were a doubly linked list, it might guarantee that every forward pointer was matched by a backwards pointer from the next Node).

So, we see here an example of where the values of some of a class's data is the responsibility of users (in which case the class needs to have get()/set() methods for that data) but the data that the class wants to control does not necessarily have get()/set() methods.

Note: the purpose of this example is not to show you how to write a linked-list class. In fact you should not "roll your own" linked-list class since you should use one of the "container classes" provided with your compiler. Ideally you'll use one of the standard container classes such as the std::list template.

Memory Allocation/Deallocation

Q: What is the difference between new/delete and malloc/free?
A: Malloc/free do not know about constructors and destructors. New and delete create and destroy objects, while malloc and free allocate and deallocate memory.

Q: What is difference between new and malloc?
A: Both malloc and new functions are used for dynamic memory allocations and the basic difference is: malloc requires a special "typecasting" when it allocates memory for eg. if the pointer used is the char pointer then after the processor allocates memory then this allocated memory needs to be typecasted to char pointer i.e (char*).but new does not requires any typecasting. Also, free is the keyword used to free the memory while using malloc and delete the keyword to free memory while using new, otherwise this will lead the memory leak.

Q: What is the difference between delete and delete[]?
A: Whenever you allocate memory with new[], you have to free the memory using delete[]. When you allocate memory with 'new', then use 'delete' without the brackets. You use new[] to allocate an array of values (always starting at the index 0).

Q: What is difference between malloc()/free() and new/delete?
A: malloc allocates memory for object in heap but doesn't invoke object's constructor to initialize the object. new allocates memory and also invokes constructor to initialize the object. malloc() and free() do not support object semantics, does not construct and destruct objects
Eg. string * ptr = (string *)(malloc (sizeof(string))) Are not safe, and does not calculate the size of the objects that it construct

The following return a pointer to void
int *p = (int *) (malloc(sizeof(int)));
int *p = new int;
Are not extensible
new and delete can be overloaded in a class
"delete" first calls the object's termination routine (i.e. its destructor) and then releases the space the object occupied on the heap memory. If an array of objects was created using new, then delete must be told that it is dealing with an array by preceding the name with an empty []:-
Int_t *my_ints = new Int_t[10];
...
delete []my_ints;

Q: What is the difference between "new" and "operator new" ?
A:"operator new" works like malloc.

Q: What is Memory alignment??
A: The term alignment primarily means the tendency of an address pointer value to be a multiple of some power of two. So a pointer with two byte alignment has a zero in the least signi_cant bit. And a pointer with four byte alignment has a zero in both the two least signi_cant bits. And so on. More alignment means a longer sequence of zero bits in the lowest bits of a pointer.

Q: Is there a way to force new to allocate memory from a specific memory area?
A: Yes. The good news is that these "memory pools" are useful in a number of situations. The bad news is that I'll have to drag you through the mire of how it works before we discuss all the uses. But if you don't know about memory pools, it might be worthwhile to slog through this FAQ you might learn something useful!

First of all, recall that a memory allocator is simply supposed to return uninitialized bits of memory; it is not supposed to produce "objects." In particular, the memory allocator is not supposed to set the virtual-pointer or any other part of the object, as that is the job of the constructor which runs after the memory allocator. Starting with a simple memory allocator function, allocate(), you would use placement new to construct an object in that memory. In other words, the following is morally equivalent to new Foo():

```
void* raw = allocate(sizeof(Foo)); // line 1
Foo* p = new(raw) Foo(); // line 2
```

Okay, assuming you've used placement new and have survived the above two lines of code, the next step is to turn your memory allocator into an object. This kind of object is called a "memory pool" or a "memory arena." This lets your users have more than one "pool" or "arena" from which memory will be allocated. Each of these memory pool objects will allocate a big chunk of memory using some specific system call (e.g., shared memory, persistent memory, stack memory, etc.; see below), and will dole it out in little chunks as needed. Your memory-pool class might look something like this:

```
class Pool {
public:
void* alloc(size_t nbytes);
void dealloc(void* p);
private:
...data members used in your pool object...
};

void* Pool::alloc(size_t nbytes)
{
...your algorithm goes here...
}

void Pool::dealloc(void* p)
{
...your algorithm goes here...
}
```

Now one of your users might have a Pool called pool, from which they could allocate objects like this:

```
Pool pool;
...
void* raw = pool.alloc(sizeof(Foo));
Foo* p = new(raw) Foo();
```
Or simply:

```
Foo* p = new(pool.alloc(sizeof(Foo))) Foo();
```
The reason it's good to turn Pool into a class is because it lets users create N different pools of memory rather than having one massive pool shared by all users. That allows users to do lots of funky things. For example, if they have a chunk of the system that allocates memory like crazy then goes away, they could allocate all their memory from a Pool, then not even bother doing any deletes on the little pieces: just deallocate the entire pool at once. Or they could set up a "shared memory" area (where the operating system specifically provides memory that is shared between multiple processes) and have the pool dole out chunks of shared memory rather than process-local memory. Another angle: many systems support a non-standard function often called alloca() which allocates a block of memory from the stack rather than the heap. Naturally this block of memory automatically goes away when the function returns, eliminating the need for explicit deletes. Someone could use alloca() to give the Pool its big chunk of memory, then all the little pieces allocated from that Pool act like they're local: they automatically vanish when the function returns. Of course the destructors don't get called in some of these cases, and if the destructors do something nontrivial you won't be able to use these techniques, but in cases where the destructor merely deallocates memory, these sorts of techniques can be useful.

Okay, assuming you survived the 6 or 8 lines of code needed to wrap your allocate function as a method of a Pool class, the next step is to change the syntax for allocating objects. The goal is to change from the rather clunky syntax new(pool.alloc(sizeof(Foo))) Foo() to the simpler syntax new(pool) Foo(). To make this happen, you need to add the following two lines of code just below the definition of your Pool class:

```
inline void* operator new(size_t nbytes, Pool& pool)
{
return pool.alloc(nbytes);
}
```
Now when the compiler sees new(pool) Foo(), it calls the above operator new and passes sizeof(Foo) and pool as parameters, and the only function that ends up using the funky pool.alloc(nbytes) method is your own operator new.

Now to the issue of how to destruct/deallocate the Foo objects. Recall that the brute force approach sometimes used with placement new is to explicitly call the destructor then explicitly deallocate the memory:

```
void sample(Pool& pool)
{
```

```
Foo* p = new(pool) Foo();
...
p->~Foo(); // explicitly call dtor
pool.dealloc(p); // explicitly release the memory
}
```
This has several problems, all of which are fixable:

The memory will leak if Foo::Foo() throws an exception.
The destruction/deallocation syntax is different from what most programmers are used to, so they'll probably screw it up.
Users must somehow remember which pool goes with which object. Since the code that allocates is often in a different function from the code that deallocates, programmers will have to pass around two pointers (a Foo* and a Pool*), which gets ugly fast (example, what if they had an array of Foos each of which potentially came from a different Pool; ugh).
We will fix them in the above order.

Problem #1: plugging the memory leak. When you use the "normal" new operator, e.g., Foo* p = new Foo(), the compiler generates some special code to handle the case when the constructor throws an exception. The actual code generated by the compiler is functionally similar to this:

```
// This is functionally what happens with Foo* p = new Foo()

Foo* p;

// don't catch exceptions thrown by the allocator itself
void* raw = operator new(sizeof(Foo));

// catch any exceptions thrown by the ctor
try {
p = new(raw) Foo(); // call the ctor with raw as this
}
catch (...) {
// oops, ctor threw an exception
operator delete(raw);
throw; // rethrow the ctor's exception
}
```
The point is that the compiler deallocates the memory if the ctor throws an exception. But in the case of the "new with parameter" syntax (commonly called "placement new"), the compiler won't know what to do if the exception occurs so by default it does nothing:

```
// This is functionally what happens with Foo* p = new(pool) Foo():

void* raw = operator new(sizeof(Foo), pool);
// the above function simply returns "pool.alloc(sizeof(Foo))"
```

```
Foo* p = new(raw) Foo();
// if the above line "throws", pool.dealloc(raw) is NOT called
```
So the goal is to force the compiler to do something similar to what it does with the global new operator. Fortunately it's simple: when the compiler sees new(pool) Foo(), it looks for a corresponding operator delete. If it finds one, it does the equivalent of wrapping the ctor call in a try block as shown above. So we would simply provide an operator delete with the following signature (be careful to get this right; if the second parameter has a different type from the second parameter of the operator new(size_t, Pool&), the compiler doesn't complain; it simply bypasses the try block when your users say new(pool) Foo()):

```
void operator delete(void* p, Pool& pool)
{
pool.dealloc(p);
}
```
After this, the compiler will automatically wrap the ctor calls of your new expressions in a try block:

```
// This is functionally what happens with Foo* p = new(pool) Foo()

Foo* p;

// don't catch exceptions thrown by the allocator itself
void* raw = operator new(sizeof(Foo), pool);
// the above simply returns "pool.alloc(sizeof(Foo))"

// catch any exceptions thrown by the ctor
try {
p = new(raw) Foo(); // call the ctor with raw as this
}
catch (...) {
// oops, ctor threw an exception
operator delete(raw, pool); // that's the magical line!!
throw; // rethrow the ctor's exception
}
```
In other words, the one-liner function operator delete(void* p, Pool& pool) causes the compiler to automagically plug the memory leak. Of course that function can be, but doesn't have to be, inline.

Problems #2 ("ugly therefore error prone") and #3 ("users must manually associate pool-pointers with the object that allocated them, which is error prone") are solved simultaneously with an additional 10-20 lines of code in one place. In other words, we add 10-20 lines of code in one place (your Pool header file) and simplify an arbitrarily large number of other places (every piece of code that uses your Pool class).

The idea is to implicitly associate a Pool* with every allocation. The Pool* associated with the global allocator would be NULL, but at least conceptually you could say every allocation has an associated Pool*. Then you replace the global operator delete so it looks up the associated Pool*, and if non-NULL, calls that Pool's deallocate function. For example, if(!) the normal deallocator used free(), the replacment for the global operator delete would look something like this:

```
void operator delete(void* p)
{
if (p != NULL) {
Pool* pool = /* somehow get the associated 'Pool*' */;
if (pool == null)
free(p);
else
pool->dealloc(p);
}
}
```

If you're not sure if the normal deallocator was free(), the easiest approach is also replace the global operator new with something that uses malloc(). The replacement for the global operator new would look something like this (note: this definition ignores a few details such as the new_handler loop and the throw std::bad_alloc() that happens if we run out of memory):

```
void* operator new(size_t nbytes)
{
if (nbytes == 0)
nbytes = 1; // so all alloc's get a distinct address
void* raw = malloc(nbytes);
...somehow associate the NULL 'Pool*' with 'raw'...
return raw;
}
```

The only remaining problem is to associate a Pool* with an allocation. One approach, used in at least one commercial product, is to use a std::map. In other words, build a look-up table whose keys are the allocation-pointer and whose values are the associated Pool*. For reasons I'll describe in a moment, it is essential that you insert a key/value pair into the map only in operator new(size_t,Pool&). In particular, you must not insert a key/value pair from the global operator new (e.g., you must not say, poolMap[p] = NULL in the global operator new). Reason: doing that would create a nasty chicken-and-egg problem since std::map probably uses the global operator new, it ends up inserting a new entry every time inserts a new entry, leading to infinite recursion bang you're dead.

Even though this technique requires a std::map look-up for each deallocation, it seems to have acceptable performance, at least in many cases.

Another approach that is faster but might use more memory and is a little trickier is to prepend a

Pool* just before all allocations. For example, if nbytes was 24, meaning the caller was asking to allocate 24 bytes, we would allocate 28 (or 32 if you think the machine requires 8-byte alignment for things like doubles and/or long longs), stuff the Pool* into the first 4 bytes, and return the pointer 4 (or 8) bytes from the beginning of what you allocated. Then your global operator delete backs off the 4 (or 8) bytes, finds the Pool*, and if NULL, uses free() otherwise calls pool->dealloc(). The parameter passed to free() and pool->dealloc() would be the pointer 4 (or 8) bytes to the left of the original parameter, p. If(!) you decide on 4 byte alignment, your code would look something like this (although as before, the following operator new code elides the usual out-of-memory handlers):

```
void* operator new(size_t nbytes)
{
if (nbytes == 0)
nbytes = 1; // so all alloc's get a distinct address
void* ans = malloc(nbytes + 4); // overallocate by 4 bytes
*(Pool**)ans = NULL; // use NULL in the global new
return (char*)ans + 4; // don't let users see the Pool*
}

void* operator new(size_t nbytes, Pool& pool)
{
if (nbytes == 0)
nbytes = 1; // so all alloc's get a distinct address
void* ans = pool.alloc(nbytes + 4); // overallocate by 4 bytes
*(Pool**)ans = &pool; // put the Pool* here
return (char*)ans + 4; // don't let users see the Pool*
}

void operator delete(void* p)
{
if (p != NULL) {
p = (char*)p - 4; // back off to the Pool*
Pool* pool = *(Pool**)p;
if (pool == null)
free(p); // note: 4 bytes left of the original p
else
pool->dealloc(p); // note: 4 bytes left of the original p
}
}
```

Naturally the last few paragraphs of this FAQ are viable only when you are allowed to change the global operator new and operator delete. If you are not allowed to change these global functions, the first three quarters of this FAQ is still applicable.

Q: How does free know the size of memory to be deleted.?
int *i = (int *)malloc(12); followed by free(i); how did free function call know how much of memory to delete?
A: It depends on the implementation, but there is usually a malloc header added to all the memory allocated through malloc. on Linux its 4 bytes of memory preceding the memory returned to you, whihc contains the number of bytes allocated + 4(itself).

so when you say,

int *i = (int *)malloc(12);

it allocates 16 bytes.

```
------------------
17 | | | |
------------------
^
|
i
```

As you can see above total of 16 bytes are allocated, first 4 bytes stores the number of bytes allocated(offset to the next memory from the start of this memory).
address of the 5th byte is returned to i. now i can access 12 bytes from this byte.

Q: How do I allocate multidimensional arrays using new?
A: There are many ways to do this, depending on how flexible you want the array sizing to be. On one extreme, if you know all the dimensions at compile-time, you can allocate multidimensional arrays statically (as in C):

```
class Fred { /*...*/ };
void someFunction(Fred& fred);

void manipulateArray()
{
const unsigned nrows = 10; // Num rows is a compile-time constant
const unsigned ncols = 20; // Num columns is a compile-time constant
Fred matrix[nrows][ncols];

for (unsigned i = 0; i < nrows; ++i) {
for (unsigned j = 0; j < ncols; ++j) {
// Here's the way you access the (i,j) element:
someFunction( matrix[i][j] );

// You can safely "return" without any special delete code:
if (today == "Tuesday" && moon.isFull())
```

```
return; // Quit early on Tuesdays when the moon is full
}
}

// No explicit delete code at the end of the function either
}
```
More commonly, the size of the matrix isn't known until run-time but you know that it will be rectangular. In this case you need to use the heap ("freestore"), but at least you are able to allocate all the elements in one freestore chunk.

```
void manipulateArray(unsigned nrows, unsigned ncols)
{
Fred* matrix = new Fred[nrows * ncols];

// Since we used a simple pointer above, we need to be VERY
// careful to avoid skipping over the delete code.
// That's why we catch all exceptions:
try {

// Here's how to access the (i,j) element:
for (unsigned i = 0; i < nrows; ++i) {
for (unsigned j = 0; j < ncols; ++j) {
someFunction( matrix[i*ncols + j] );
}
}

// If you want to quit early on Tuesdays when the moon is full,
// make sure to do the delete along ALL return paths:
if (today == "Tuesday" && moon.isFull()) {
delete[] matrix;
return;
}

...insert code here to fiddle with the matrix...

}
catch (...) {
// Make sure to do the delete when an exception is thrown:
delete[] matrix;
throw; // Re-throw the current exception
}

// Make sure to do the delete at the end of the function too:
delete[] matrix;
}
```

Finally at the other extreme, you may not even be guaranteed that the matrix is rectangular. For example, if each row could have a different length, you'll need to allocate each row individually. In the following function, ncols[i] is the number of columns in row number i, where i varies between 0 and nrows-1 inclusive.

```
void manipulateArray(unsigned nrows, unsigned ncols[])
{
typedef Fred* FredPtr;

// There will not be a leak if the following throws an exception:
FredPtr* matrix = new FredPtr[nrows];

// Set each element to NULL in case there is an exception later.
// (See comments at the top of the try block for rationale.)
for (unsigned i = 0; i < nrows; ++i)
matrix[i] = NULL;

// Since we used a simple pointer above, we need to be
// VERY careful to avoid skipping over the delete code.
// That's why we catch all exceptions:
try {

// Next we populate the array. If one of these throws, all
// the allocated elements will be deleted (see catch below).
for (unsigned i = 0; i < nrows; ++i)
matrix[i] = new Fred[ ncols[i] ];

// Here's how to access the (i,j) element:
for (unsigned i = 0; i < nrows; ++i) {
for (unsigned j = 0; j < ncols[i]; ++j) {
someFunction( matrix[i][j] );
}
}

// If you want to quit early on Tuesdays when the moon is full,
// make sure to do the delete along ALL return paths:
if (today == "Tuesday" && moon.isFull()) {
for (unsigned i = nrows; i > 0; --i)
delete[] matrix[i-1];
delete[] matrix;
return;
}

...insert code here to fiddle with the matrix...
```

```
}
catch (...) {
// Make sure to do the delete when an exception is thrown:
// Note that some of these matrix[...] pointers might be
// NULL, but that's okay since it's legal to delete NULL.
for (unsigned i = nrows; i > 0; --i)
delete[] matrix[i-1];
delete[] matrix;
throw; // Re-throw the current exception
}

// Make sure to do the delete at the end of the function too.
// Note that deletion is the opposite order of allocation:
for (unsigned i = nrows; i > 0; --i)
delete[] matrix[i-1];
delete[] matrix;
}
```

Note the funny use of matrix[i-1] in the deletion process. This prevents wrap-around of the unsigned value when i goes one step below zero.

Finally, note that pointers and arrays are evil. It is normally much better to encapsulate your pointers in a class that has a safe and simple interface. The following FAQ shows how to do this.

Q: Can I free() pointers allocated with new? Can I delete pointers allocated with malloc()?
A: No!

It is perfectly legal, moral, and wholesome to use malloc() and delete in the same program, or to use new and free() in the same program. But it is illegal, immoral, and despicable to call free() with a pointer allocated via new, or to call delete on a pointer allocated via malloc().

Beware! I occasionally get e-mail from people telling me that it works OK for them on machine X and compiler Y. Just because they don't see bad symptoms in a simple test case doesn't mean it won't crash in the field. Even if they know it won't crash on their particular compiler doesn't mean it will work safely on another compiler, another platform, or even another version of the same compiler.

Beware! Sometimes people say, "But I'm just working with an array of char." Nonetheless do not mix malloc() and delete on the same pointer, or new and free() on the same pointer! If you allocated via p = new char[n], you must use delete[] p; you must not use free(p). Or if you allocated via p = malloc(n), you must use free(p); you must not use delete[] p or delete p! Mixing these up could cause a catastrophic failure at runtime if the code was ported to a new machine, a new compiler, or even a new version of the same compiler.

You have been warned.

Q: Why should I use new instead of trustworthy old malloc()?

A: Constructors/destructors, type safety, overridability.

Constructors/destructors: unlike malloc(sizeof(Fred)), new Fred() calls Fred's constructor. Similarly, delete p calls *p's destructor.
Type safety: malloc() returns a void* which isn't type safe. new Fred() returns a pointer of the right type (a Fred*).
Overridability: new is an operator that can be overridden by a class, while malloc() is not overridable on a per-class basis.

Q: Can I use realloc() on pointers allocated via new?
A: No!

When realloc() has to copy the allocation, it uses a bitwise copy operation, which will tear many C++ objects to shreds. C++ objects should be allowed to copy themselves. They use their own copy constructor or assignment operator.

Besides all that, the heap that new uses may not be the same as the heap that malloc() and realloc() use!

Q: Do I need to check for NULL after p = new Fred()?
A: No! (But if you have an old compiler, you may have to force the new operator to throw an exception if it runs out of memory.)

It turns out to be a real pain to always write explicit NULL tests after every new allocation. Code like the following is very tedious:

```
Fred* p = new Fred();
if (p == NULL)
throw std::bad_alloc();
```

If your compiler doesn't support (or if you refuse to use) exceptions, your code might be even more tedious:

```
Fred* p = new Fred();
if (p == NULL) {
std::cerr << "Couldn't allocate memory for a Fred" << std::endl;
abort();
}
```

Take heart. In C++, if the runtime system cannot allocate sizeof(Fred) bytes of memory during p = new Fred(), a std::bad_alloc exception will be thrown. Unlike malloc(), new never returns NULL!

Therefore you should simply write:

Fred* p = new Fred(); // No need to check if p is NULL
However, if your compiler is old, it may not yet support this. Find out by checking your compiler's documentation under "new". If you have an old compiler, you may have to force the compiler to have this behavior.

Note: If you are using Microsoft Visual C++, to get new to throw an exception when it fails you must #include some standard header in at least one of your .cpp files. For example, you could #include (or or or ...).

Q: How can I convince my (older) compiler to automatically check new to see if it returns NULL?
A: Eventually your compiler will.

If you have an old compiler that doesn't automagically perform the NULL test, you can force the runtime system to do the test by installing a "new handler" function. Your "new handler" function can do anything you want, such as throw an exception, delete some objects and return (in which case operator new will retry the allocation), print a message and abort() the program, etc.

Here's a sample "new handler" that prints a message and throws an exception. The handler is installed using std::set_new_handler():

```
#include // To get std::set_new_handler
#include // To get abort()
#include // To get std::cerr

class alloc_error : public std::exception {
public:
alloc_error() : exception() { }
};

void myNewHandler()
{
// This is your own handler. It can do anything you want.
throw alloc_error();
}

int main()
{
std::set_new_handler(myNewHandler); // Install your "new handler"
...
}
```

After the std::set_new_handler() line is executed, operator new will call your myNewHandler() if/when it runs out of memory. This means that new will never return NULL:

Fred* p = new Fred(); // No need to check if p is NULL
Note: If your compiler doesn't support exception handling, you can, as a last resort, change the line throw ...; to:

std::cerr << "Attempt to allocate memory failed!" << std::endl;
abort();
Note: If some global/static object's constructor uses new, it won't use the myNewHandler() function since that constructor will get called before main() begins. Unfortunately there's no convenient way to guarantee that the std::set_new_handler() will be called before the first use of new. For example, even if you put the std::set_new_handler() call in the constructor of a global object, you still don't know if the module ("compilation unit") that contains that global object will be elaborated first or last or somewhere inbetween. Therefore you still don't have any guarantee that your call of std::set_new_handler() will happen before any other global's constructor gets invoked.

Q: Do I need to check for NULL before delete p?
A: No!

The C++ language guarantees that delete p will do nothing if p is equal to NULL. Since you might get the test backwards, and since most testing methodologies force you to explicitly test every branch point, you should not put in the redundant if test.

Wrong:

if (p != NULL)
delete p;

Right:

delete p;

Q: What are the two steps that happen when I say delete p?
A: N delete p is a two-step process: it calls the destructor, then releases the memory. The code generated for delete p is functionally similar to this (assuming p is of type Fred*):

// Original code: delete p;
if (p != NULL) {
p->~Fred();
operator delete(p);
}
The statement p->~Fred() calls the destructor for the Fred object pointed to by p.

The statement operator delete(p) calls the memory deallocation primitive, void operator

delete(void* p). This primitive is similar in spirit to free(void* p). (Note, however, that these two are not interchangeable; e.g., there is no guarantee that the two memory deallocation primitives even use the same heap!)

Q: In p = new Fred(), does the Fred memory "leak" if the Fred constructor throws an exception?
A: No.

If an exception occurs during the Fred constructor of p = new Fred(), the C++ language guarantees that the memory sizeof(Fred) bytes that were allocated will automagically be released back to the heap.

Here are the details: new Fred() is a two-step process:

sizeof(Fred) bytes of memory are allocated using the primitive void* operator new(size_t nbytes). This primitive is similar in spirit to malloc(size_t nbytes). (Note, however, that these two are not interchangeable; e.g., there is no guarantee that the two memory allocation primitives even use the same heap!).
It constructs an object in that memory by calling the Fred constructor. The pointer returned from the first step is passed as the this parameter to the constructor. This step is wrapped in a try ... catch block to handle the case when an exception is thrown during this step.
Thus the actual generated code is functionally similar to:

```
// Original code: Fred* p = new Fred();
Fred* p = (Fred*) operator new(sizeof(Fred));
try {
new(p) Fred(); // Placement new
}
catch (...) {
operator delete(p); // Deallocate the memory
throw; // Re-throw the exception
}
```

The statement marked "Placement new" calls the Fred constructor. The pointer p becomes the this pointer inside the constructor, Fred::Fred().

Q: How do I allocate / unallocate an array of things?
A: Use p = new T[n] and delete[] p:

```
Fred* p = new Fred[100];
...
delete[] p;
```

Any time you allocate an array of objects via new (usually with the [n] in the new expression), you must use [] in the delete statement. This syntax is necessary because there is no syntactic difference between a pointer to a thing and a pointer to an array of things (something we inherited from C).

Q: What if I forget the [] when deleteing array allocated via new T[n]?
A: All life comes to a catastrophic end.

It is the programmer's not the compiler's responsibility to get the connection between new T[n] and delete[] p correct. If you get it wrong, neither a compile-time nor a run-time error message will be generated by the compiler. Heap corruption is a likely result. Or worse. Your program will probably die.

Q: Can I drop the [] when deleteing array of some built-in type (char, int, etc)?
A: No!

Sometimes programmers think that the [] in the delete[] p only exists so the compiler will call the appropriate destructors for all elements in the array. Because of this reasoning, they assume that an array of some built-in type such as char or int can be deleted without the []. E.g., they assume the following is valid code:

```
void userCode(int n)
{
char* p = new char[n];
...
delete p; //  ERROR! Should be delete[] p !
}
```

But the above code is wrong, and it can cause a disaster at runtime. In particular, the code that's called for delete p is operator delete(void*), but the code that's called for delete[] p is operator delete[](void*). The default behavior for the latter is to call the former, but users are allowed to replace the latter with a different behavior (in which case they would normally also replace the corresponding new code in operator new[](size_t)). If they replaced the delete[] code so it wasn't compatible with the delete code, and you called the wrong one (i.e., if you said delete p rather than delete[] p), you could end up with a disaster at runtime.

Q: After p = new Fred[n], how does the compiler know there are n objects to be destructed during delete[] p?
A: Short answer: Magic.

Long answer: The run-time system stores the number of objects, n, somewhere where it can be retrieved if you only know the pointer, p. There are two popular techniques that do this. Both these techniques are in use by commercial-grade compilers, both have tradeoffs, and neither is perfect. These techniques are:

Over-allocate the array and put n just to the left of the first Fred object.
Use an associative array with p as the key and n as the value.

Q: Is it legal (and moral) for a member function to say delete this?
A: As long as you're careful, it's OK for an object to commit suicide (delete this).

Here's how I define "careful":

You must be absolutely 100% positive sure that this object was allocated via new (not by new[], nor by placement new, nor a local object on the stack, nor a global, nor a member of another object; but by plain ordinary new).
You must be absolutely 100% positive sure that your member function will be the last member function invoked on this object.
You must be absolutely 100% positive sure that the rest of your member function (after the delete this line) doesn't touch any piece of this object (including calling any other member functions or touching any data members).
You must be absolutely 100% positive sure that no one even touches the this pointer itself after the delete this line. In other words, you must not examine it, compare it with another pointer, compare it with NULL, print it, cast it, do anything with it.
Naturally the usual caveats apply in cases where your this pointer is a pointer to a base class when you don't have a virtual destructor.

Exception Handling

Q: What happens when a function throws an exception that was not specified by an exception specification for this function?
A: Unexpected() is called, which, by default, will eventually trigger abort().

Q: What does throw; (without an exception object after the throw keyword) mean? Where would I use it?
A: You might see code that looks something like this:

```
class MyException {
public:
...
void addInfo(const std::string& info);
...
};

void f()
{
try {
...
}
catch (MyException& e) {
e.addInfo("f() failed");
throw;
}
}
```

In this example, the statement throw; means "re-throw the current exception." Here, a function caught an exception (by non-const reference), modified the exception (by adding information to it), and then re-threw the exception. This idiom can be used to implement a simple form of stack-trace, by adding appropriate catch clauses in the important functions of your program.

Another re-throwing idiom is the "exception dispatcher":

```
void handleException()
{
try {
throw;
}
catch (MyException& e) {
...code to handle MyException...
}
catch (YourException& e) {
...code to handle YourException...
}
```

```
}

void f()
{
try {
...something that might throw...
}
catch (...) {
handleException();
}
}
```

This idiom allows a single function (handleException()) to be re-used to handle exceptions in a number of other functions.

Q: How do I throw polymorphically?
A: Sometimes people write code like:

```
class MyExceptionBase { };

class MyExceptionDerived : public MyExceptionBase { };

void f(MyExceptionBase& e)
{
// ...
throw e;
}

void g()
{
MyExceptionDerived e;
try {
f(e);
}
catch (MyExceptionDerived& e) {
...code to handle MyExceptionDerived...
}
catch (...) {
...code to handle other exceptions...
}
}
```

If you try this, you might be surprised at run-time when your catch (...) clause is entered, and not your catch (MyExceptionDerived&) clause. This happens because you didn't throw polymorphically. In function f(), the statement throw e; throws an object with the same type as the static type of the expression e. In other words, it throws an instance of MyExceptionBase. The throw statement behaves as-if the thrown object is copied, as opposed to making a "virtual

copy".

Fortunately it's relatively easy to correct:

```
class MyExceptionBase {
public:
virtual void raise();
};

void MyExceptionBase::raise()
{ throw *this; }

class MyExceptionDerived : public MyExceptionBase {
public:
virtual void raise();
};

void MyExceptionDerived::raise()
{ throw *this; }

void f(MyExceptionBase& e)
{
// ...
e.raise();
}

void g()
{
MyExceptionDerived e;
try {
f(e);
}
catch (MyExceptionDerived& e) {
...code to handle MyExceptionDerived...
}
catch (...) {
...code to handle other exceptions...
}
}
```

Note that the throw statement has been moved into a virtual function. The statement e.raise() will exhibit polymorphic behavior, since raise() is declared virtual and e was passed by reference. As before, the thrown object will be of the static type of the argument in the throw statement, but within MyExceptionDerived::raise(), that static type is MyExceptionDerived, not MyExceptionBase.

Q: When I throw this object, how many times will it be copied?
A: Depends. Might be "zero."

Objects that are thrown must have a publicly accessible copy-constructor. The compiler is allowed to generate code that copies the thrown object any number of times, including zero. However even if the compiler never actually copies the thrown object, it must make sure the exception class's copy constructor exists and is accessible.

Q: But MFC seems to encourage the use of catch-by-pointer; should I do the same?
A: Depends. If you're using MFC and catching one of their exceptions, by all means, do it their way. Same goes for any framework: when in Rome, do as the Romans. Don't try to force a framework into your way of thinking, even if "your" way of thinking is "better." If you decide to use a framework, embrace its way of thinking use the idioms that its authors expected you to use.

But if you're creating your own framework and/or a piece of the system that does not directly depend on MFC, then don't catch by pointer just because MFC does it that way. When you're not in Rome, you don't necessarily do as the Romans. In this case, you should not. Libraries like MFC predated the standardization of exception handling in the C++ language, and some of these libraries use a backwards-compatible form of exception handling that requires (or at least encourages) you to catch by pointer.

The problem with catching by pointer is that it's not clear who (if anyone) is responsible for deleting the pointed-to object. For example, consider the following:

```
MyException x;

void f()
{
MyException y;

try {
switch (rand() % 3) {
case 0: throw new MyException;
case 1: throw &x;
case 2: throw &y;
}
}
catch (MyException* p) {
...  should we delete p here or not???!?
}
}
```

There are three basic problems here:

It might be tough to decide whether to delete p within the catch clause. For example, if object x is inaccessible to the scope of the catch clause, such as when it's buried in the private part of

some class or is static within some other compilation unit, it might be tough to figure out what to do.
If you solve the first problem by consistently using new in the throw (and therefore consistently using delete in the catch), then exceptions always use the heap which can cause problems when the exception was thrown because the system was running low on memory.
If you solve the first problem by consistently not using new in the throw (and therefore consistently not using delete in the catch), then you probably won't be able to allocate your exception objects as locals (since then they might get destructed too early), in which case you'll have to worry about thread-safety, locks, semaphores, etc. (static objects are not intrinsically thread-safe).
This isn't to say it's not possible to work through these issues. The point is simply this: if you catch by reference rather than by pointer, life is easier. Why make life hard when you don't have to?

The moral: avoid throwing pointer expressions, and avoid catching by pointer, unless you're using an existing library that "wants" you to do so.

Q: What are some ways try / catch / throw can improve software quality?
A: By eliminating one of the reasons for if statements.

The commonly used alternative to try / catch / throw is to return a return code (sometimes called an error code) that the caller explicitly tests via some conditional statement such as if. For example, printf(), scanf() and malloc() work this way: the caller is supposed to test the return value to see if the function succeeded.

Although the return code technique is sometimes the most appropriate error handling technique, there are some nasty side effects to adding unnecessary if statements:

Degrade quality: It is well known that conditional statements are approximately ten times more likely to contain errors than any other kind of statement. So all other things being equal, if you can eliminate conditionals / conditional statements from your code, you will likely have more robust code.
Slow down time-to-market: Since conditional statements are branch points which are related to the number of test cases that are needed for white-box testing, unnecessary conditional statements increase the amount of time that needs to be devoted to testing. Basically if you don't exercise every branch point, there will be instructions in your code that will never have been executed under test conditions until they are seen by your users/customers. That's bad.
Increase development cost: Bug finding, bug fixing, and testing are all increased by unnecessary control flow complexity.
So compared to error reporting via return-codes and if, using try / catch / throw is likely to result in code that has fewer bugs, is less expensive to develop, and has faster time-to-market. Of course if your organization doesn't have any experiential knowledge of try / catch / throw, you might want to use it on a toy project first just to make sure you know what you're doing you should always get used to a weapon on the firing range before you bring it to the front lines of a shooting war.

Q: What should I throw?
A: C++, unlike just about every other language with exceptions, is very accomodating when it comes to what you can throw. In fact, you can throw anything you like. That begs the question then, what should you throw?

Generally, it's best to throw objects, not built-ins. If possible, you should throw instances of classes that derive (ultimately) from the std::exception class. By making your exception class inherit (ultimately) from the standard exception base-class, you are making life easier for your users (they have the option of catching most things via std::exception), plus you are probably providing them with more information (such as the fact that your particular exception might be a refinement of std::runtime_error or whatever).

The most common practice is to throw a temporary:

```
#include

class MyException : public std::runtime_error {
public:
MyException() : std::runtime_error("MyException") { }
};

void f()
{
// ...
throw MyException();
}
```

Here, a temporary of type MyException is created and thrown. Class MyException inherits from class std::runtime_error which (ultimately) inherits from class std::exception.

Q: What should I catch?
A: In keeping with the C++ tradition of "there's more than one way to do that" (translation: "give programmers options and tradeoffs so they can decide what's best for them in their situation"), C++ allows you a variety of options for catching.

You can catch by value.
You can catch by reference.
You can catch by pointer.
In fact, you have all the flexibility that you have in declaring function parameters, and the rules for whether a particular exception matches (i.e., will be caught by) a particular catch clause are almost exactly the same as the rules for parameter compatibility when calling a function.

Given all this flexibility, how do you decide what to catch? Simple: unless there's a good reason not to, catch by reference. Avoid catching by value, since that causes a copy to be made and the

copy can have different behavior from what was thrown. Only under very special circumstances should you catch by pointer.

Pointers

Q: What is a dangling pointer?
A: 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.

Q: What is Memory Leak?
A: Memory which has no pointer pointing to it and there is no way to delete or reuse this memory(object), it causes Memory leak.

```
{
Base *b = new base();
}
```

Out of this scope b no longer exists, but the memory it was pointing to was not deleted. Pointer b itself was destroyed when it went out of scope.

Q: What is auto pointer?
A: The simplest example of a smart pointer is auto_ptr, which is included in the standard C++ library. Auto Pointer only takes care of Memory leak and does nothing about dangling pointers issue. You can find it in the header . Here is part of auto_ptr's implementation, to illustrate what it does:

```
template class auto_ptr
{
T* ptr;
public:
explicit auto_ptr(T* p = 0) : ptr(p) {}
~auto_ptr() {delete ptr;}
T& operator*() {return *ptr;}
T* operator->() {return ptr;}
// ...
};
```

As you can see, auto_ptr is a simple wrapper around a regular pointer. It forwards all meaningful operations to this pointer (dereferencing and indirection). Its smartness in the destructor: the destructor takes care of deleting the pointer.
For the user of auto_ptr, this means that instead of writing:

```
void foo()
{
MyClass* p(new MyClass);
p->DoSomething();
```

```
delete p;
}
```

You can write:

```
void foo()
{
auto_ptr p(new MyClass);
p->DoSomething();
}
```

And trust p to cleanup after itself.

Q: What issue do auto_ptr objects address?
A: If you use auto_ptr objects you would not have to be concerned with heap objects not being deleted even if the exception is thrown.

Q: What is a smart pointer?
A: A smart pointer is a C++ class that mimics a regular pointer in syntax and some semantics, but it does more. Because smart pointers to different types of objects tend to have a lot of code in common, almost all good-quality smart pointers in existence are templated by the pointee type, as you can see in the following code:

```
template
class SmartPtr
{
public:
explicit SmartPtr(T* pointee) : pointee_(pointee);
SmartPtr& operator=(const SmartPtr& other);
~SmartPtr();
T& operator*() const
{
...
return *pointee_;
}
T* operator->() const
{
...
return pointee_;
}
private:
T* pointee_;
...
};
```

SmartPtr aggregates a pointer to T in its member variable pointee_. That's what most smart pointers do. In some cases, a smart pointer might aggregate some handles to data and compute the pointer on the fly.
The two operators give SmartPtr pointer-like syntax and semantics. That is, you can write

```
class Widget
{
public:
void Fun();
};

SmartPtr sp(new Widget);
sp->Fun();
(*sp).Fun();
```

Aside from the definition of sp, nothing reveals it as not being a pointer. This is the mantra of smart pointers: You can replace pointer definitions with smart pointer definitions without incurring major changes to your application's code. You thus get extra goodies with ease. Minimizing code changes is very appealing and vital for getting large applications to use smart pointers. As you will soon see, however, smart pointers are not a free lunch.

Q: Is there any problem with the following : char*a=NULL; char& p = *a;?
A: The result is undefined. You should never do this. A reference must always refer to some object.

Q: What is the difference between a pointer and a reference?
A: A reference must always refer to some object and, therefore, must always be initialized; pointers do not have such restrictions. A pointer can be reassigned to point to different objects while a reference always refers to an object with which it was initialized.

Q: What is the difference between const char *myPointer and char *const myPointer?
A: Const char *myPointer is a non constant pointer to constant data; while char *const myPointer is a constant pointer to non constant data.

Q: When should I use references, and when should I use pointers?
A: Use references when you can, and pointers when you have to.

References are usually preferred over pointers whenever you don't need "reseating". This usually means that references are most useful in a class's public interface. References typically appear on the skin of an object, and pointers on the inside.

The exception to the above is where a function's parameter or return value needs a "sentinel" reference a reference that does not refer to an object. This is usually best done by

returning/taking a pointer, and giving the NULL pointer this special significance (references should always alias objects, not a dereferenced NULL pointer).

Note: Old line C programmers sometimes don't like references since they provide reference semantics that isn't explicit in the caller's code. After some C++ experience, however, one quickly realizes this is a form of information hiding, which is an asset rather than a liability. E.g., programmers should write code in the language of the problem rather than the language of the machine.

Strings

Q: What happens if you write this code?

```
string& foo()
{
return "Hello World";
}

cout << foo() << endl;
```

A: 1. Will give an error since Hello World is created as a unnamed character pointer to const. it is being assigned to non-const reference which is not allowed.
could not convert `'"Hello World"' to `std::string&'

```
2. const string& foo1()
{
return "Hello World";
}
```

Gives a warning, since you are returning a reference to temporary, which will die immediately when the expression is completed.
classsize.C:7: warning: returning reference to temporary
output : Aborted. Segment fault.

```
3. Char *foo1()
{
return “Hello World”;
}
```

Returning the address of character literal which is created on the static memory.

In C++, the compiler allows the use of string literals to initialize character arrays. A string literal consists of zero or more characters surrounded by double quotation marks ("). A string literal represents a sequence of characters that, taken together, form a null-terminated string. The compiler creates static storage space for the string, null-terminates it, and puts the address of this space into the char* variable. The type of a literal string is an array of const chars.

```
char* szMyString = "Hello world.";
szMyString[3] = 'q'; // undefined, modifying static buffer!!!
```

In the following example, the compiler automatically puts a null-character at the end of the literal string of characters "Hello world". It then creates a storage space for the resulting string - this is an array of const chars. Then it puts the starting address of this array into the szMyString variable. We will try to modify this string (wherever it is stored) by accessing it via an index into szMyString. This is a Bad Thing; the standard does not say where the compiler puts literal

strings. They can go anywhere, possibly in some place in memory that you shouldn't be modifying.

Q: How do I convert an integer to a string?
A: The simplest way is to use a stringstream:

```
#include
#include
#include
using namespace std;

string itos(int i) // convert int to string
{
stringstream s;
s << i;
return s.str();
}

int main()
{
int i = 127;
string ss = itos(i);
const char* p = ss.c_str();

cout << ss << " " << p << "\n";
}
```

Naturally, this technique works for converting any type that you can output using << to a string.

C Vs. C++

Q: How do you link a C++ program to C functions?
A: By using the extern "C" linkage specification around the C function declarations. Programmers should know about mangled function names and type-safe linkages. Then they should explain how the extern "C" linkage specification statement turns that feature off during compilation so that the linker properly links function calls to C functions.

Q: Is there anything you can do in C++ that you cannot do in C?
A: No. There is nothing you can do in C++ that you cannot do in C. After all you can write a C++ compiler in C

Q: What are the differences between a struct in C and in C++?
A: In C++ a struct is similar to a class except for the default access specifier(refere to other question in the document). In C we have to include the struct keyword when declaring struct. In c++ we don't have to.

Q: What does extern "C" int func(int *, Foo) accomplish?
A: It will turn o_ "name mangling" for func so that one can link to code compiled by a C compiler.

Q: What are the access privileges in C++? What is the default access level?
A: The access privileges in C++ are private, public and protected. The default access level assigned to members of a class is private. Private members of a class are accessible only within the class and by friends of the class. Protected members are accessible by the class itself and it's sub-classes. Public members of a class can be accessed by anyone.

Q: How does C++ help with the tradeoff of safety vs. usability?
A: In C, encapsulation was accomplished by making things static in a compilation unit or module. This prevented another module from accessing the static stuff. (By the way, static data at file-scope is now deprecated in C++: don't do that.)

Unfortunately this approach doesn't support multiple instances of the data, since there is no direct support for making multiple instances of a module's static data. If multiple instances were needed in C, programmers typically used a struct. But unfortunately C structs don't support encapsulation. This exacerbates the tradeoff between safety (information hiding) and usability (multiple instances).

In C++, you can have both multiple instances and encapsulation via a class. The public part of a class contains the class's interface, which normally consists of the class's public member functions and its friend functions. The private and/or protected parts of a class contain the class's implementation, which is typically where the data lives.

The end result is like an "encapsulated struct." This reduces the tradeoff between safety (information hiding) and usability (multiple instances).

Overloading Operators

Q: Name the operators that cannot be overloaded?
A:sizeof, ., .*, .->, ::, ?:

Q: What is overloading??
A: With the C++ language, you can overload functions and operators. Overloading is the practice of supplying more than one definition for a given function name in the same scope.
- Any two functions in a set of overloaded functions must have different argument lists.
- Overloading functions with argument lists of the same types, based on return type alone, is an error.

Q: How are prefix and postfix versions of operator++() differentiated?
A: The postfix version of operator++() has a dummy parameter of type int. The prefix version does not have dummy parameter.

Q: Can you overload a function based only on whether a parameter is a value or a reference?
A: No. Passing by value and by reference looks identical to the caller.

Q: What's the deal with operator overloading?
A: It allows you to provide an intuitive interface to users of your class, plus makes it possible for templates to work equally well with classes and built-in/intrinsic types.

Operator overloading allows C/C++ operators to have user-defined meanings on user-defined types (classes). Overloaded operators are syntactic sugar for function calls:

```
class Fred {
public:
...
};

#if 0

// Without operator overloading:
Fred add(const Fred& x, const Fred& y);
Fred mul(const Fred& x, const Fred& y);

Fred f(const Fred& a, const Fred& b, const Fred& c)
{
return add(add(mul(a,b), mul(b,c)), mul(c,a)); // Yuk...
}

#else

// With operator overloading:
```

```
Fred operator+ (const Fred& x, const Fred& y);
Fred operator* (const Fred& x, const Fred& y);

Fred f(const Fred& a, const Fred& b, const Fred& c)
{
return a*b + b*c + c*a;
}

#endif
```

Q: What are the benefits of operator overloading?
A: By overloading standard operators on a class, you can exploit the intuition of the users of that class. This lets users program in the language of the problem domain rather than in the language of the machine.

The ultimate goal is to reduce both the learning curve and the defect rate.

Q: What are some examples of operator overloading?
A: Here are a few of the many examples of operator overloading:

myString + yourString might concatenate two std::string objects
myDate++ might increment a Date object
a * b might multiply two Number objects
a[i] might access an element of an Array object
x = *p might dereference a "smart pointer" that "points" to a disk record – it could seek to the location on disk where p "points" and return the appropriate record into x

Q: But operator overloading makes my class look ugly; isn't it supposed to make my code clearer?
A: Operator overloading makes life easier for the users of a class, not for the developer of the class!

Consider the following example.

```
class Array {
public:
int& operator[] (unsigned i); // Some people don't like this syntax
...
};

inline
int& Array::operator[] (unsigned i) // Some people don't like this syntax
{
...
}
```

Some people don't like the keyword operator or the somewhat funny syntax that goes with it in the body of the class itself. But the operator overloading syntax isn't supposed to make life easier for the developer of a class. It's supposed to make life easier for the users of the class:

```
int main()
{
Array a;
a[3] = 4; // User code should be obvious and easy to understand...
...
}
```

Remember: in a reuse-oriented world, there will usually be many people who use your class, but there is only one person who builds it (yourself); therefore you should do things that favor the many rather than the few.

Q: What operators can/cannot be overloaded?
A: Most can be overloaded. The only C operators that can't be are . and ?: (and sizeof, which is technically an operator). C++ adds a few of its own operators, most of which can be overloaded except :: and .*.

Here's an example of the subscript operator (it returns a reference). First without operator overloading:

```
class Array {
public:
int& elem(unsigned i) { if (i > 99) error(); return data[i]; }
private:
int data[100];
};

int main()
{
Array a;
a.elem(10) = 42;
a.elem(12) += a.elem(13);
...
}
```

Now the same logic is presented with operator overloading:

```
class Array {
public:
int& operator[] (unsigned i) { if (i > 99) error(); return data[i]; }
private:
int data[100];
```

```
};

int main()
{
Array a;
a[10] = 42;
a[12] += a[13];
...
}
```

Q: Can I overload operator== so it lets me compare two char[] using a string comparison?
A: No: at least one operand of any overloaded operator must be of some user-defined type (most of the time that means a class).

But even if C++ allowed you to do this, which it doesn't, you wouldn't want to do it anyway since you really should be using a std::string-like class rather than an array of char in the first place since arrays are evil.

Q: Can I create a operator** for "to-the-power-of" operations?
A: Nope.

The names of, precedence of, associativity of, and arity of operators is fixed by the language. There is no operator** in C++, so you cannot create one for a class type.

If you're in doubt, consider that x ** y is the same as x * (*y) (in other words, the compiler assumes y is a pointer). Besides, operator overloading is just syntactic sugar for function calls. Although this particular syntactic sugar can be very sweet, it doesn't add anything fundamental. I suggest you overload pow(base,exponent) (a double precision version is in).

By the way, operator^ can work for to-the-power-of, except it has the wrong precedence and associativity.

Q: Okay, that tells me the operators I can override; which operators should I override?
A: Bottom line: don't confuse your users.

Remember the purpose of operator overloading: to reduce the cost and defect rate in code that uses your class. If you create operators that confuse your users (because they're cool, because they make the code faster, because you need to prove to yourself that you can do it; doesn't really matter why), you've violated the whole reason for using operator overloading in the first place.

Q: What are some guidelines / "rules of thumb" for overloading operators?
A: Here are a few guidelines / rules of thumb (but be sure to read the previous FAQ before reading this list):

Use common sense. If your overloaded operator makes life easier and safer for your users, do it; otherwise don't. This is the most important guideline. In fact it is, in a very real sense, the only

guideline; the rest are just special cases.
If you define arithmetic operators, maintain the usual arithmetic identities. For example, if your class defines x + y and x - y, then x + y - y ought to return an object that is behaviorally equivalent to x. The term behaviorally equivalent is defined in the bullet on x == y below, but simply put, it means the two objects should ideally act like they have the same state. This should be true even if you decide not to define an == operator for objects of your class.
You should provide arithmetic operators only when they make logical sense to users. Subtracting two dates makes sense, logically returning the duration between those dates, so you might want to allow date1 - date2 for objects of your Date class (provided you have a reasonable class/type to represent the duration between two Date objects). However adding two dates makes no sense: what does it mean to add July 4, 1776 to June 5, 1959? Similarly it makes no sense to multiply or divide dates, so you should not define any of those operators.
You should provide mixed-mode arithmetic operators only when they make logical sense to users. For example, it makes sense to add a duration (say 35 days) to a date (say July 4, 1776), so you might define date + duration to return a Date. Similarly date - duration could also return a Date. But duration - date does not make sense at the conceptual level (what does it mean to subtract July 4, 1776 from 35 days?) so you should not define that operator.
If you provide constructive operators, they should return their result by value. For example, x + y should return its result by value. If it returns by reference, you will probably run into lots of problems figuring out who owns the referent and when the referent will get destructed. Doesn't matter if returning by reference is more efficient; it is probably wrong. See the next bullet for more on this point.
If you provide constructive operators, they should not change their operands. For example, x + y should not change x. For some crazy reason, programmers often define x + y to be logically the same as x += y because the latter is faster. But remember, your users expect x + y to make a copy. In fact they selected the + operator (over, say, the += operator) precisely because they wanted a copy. If they wanted to modify x, they would have used whatever is equivalent to x += y instead. Don't make semantic decisions for your users; it's their decision, not yours, whether they want the semantics of x + y vs. x += y. Tell them that one is faster if you want, but then step back and let them make the final decision they know what they're trying to achieve and you do not.
If you provide constructive operators, they should allow promotion of the left-hand operand. For example, if your class Fraction supports promotion from int to Fraction (via the non-explicit ctor Fraction::Fraction(int)), and if you allow x - y for two Fraction objects, you should also allow 42 - y. In practice that simply means that your operator-() should not be a member function of Fraction. Typically you will make it a friend, if for no other reason than to force it into the public: part of the class, but even if it is not a friend, it should not be a member.
In general, your operator should change its operand(s) if and only if the operands get changed when you apply the same operator to intrinsic types. x == y and x << y should not change either operand; x *= y and x <<= y should (but only the left-hand operand).
If you define x++ and ++x, maintain the usual identities. For example, x++ and ++x should have should have the same observable effect on x, and should differ only in what they return. ++x should return x by reference; x++ should either return a copy (by value) of the original state of x or should have a void return-type. You're usually better off returning a copy of the original state of x by value, especially if your class will be used in generic algorithms. The easy way to do that is to implement x++ using three lines: make a local copy of *this, call ++x (i.e., this-

>operator++()), then return the local copy. Similar comments for x-- and --x.
If you define ++x and x += 1, maintain the usual identities. For example, these expressions should have the same observable behavior, including the same result. Among other things, that means your += operator should return x by reference. Similar comments for --x and x -= 1.
If you define *p and p[0] for pointer-like objects, maintain the usual identities. For example, these two expressions should have the same result and neither should change p.
If you define p[i] and *(p+i) for pointer-like objects, maintain the usual identities. For example, these two expressions should have the same result and neither should change p. Similar comments for p[-i] and *(p-i).
Subscript operators generally come in pairs; see on const-overloading.
If you define x == y, then x == y should be true if and only if the two objects are behaviorally equivalent. In this bullet, the term "behaviorally equivalent" means the observable behavior of any operation or sequence of operations applied to x will be the same as when applied to y. The term "operation" means methods, friends, operators, or just about anything else you can do with these objects (except, of course, the address-of operator). You won't always be able to achieve that goal, but you ought to get close, and you ought to document any variances (other than the address-of operator).
If you define x == y and x = y, maintain the usual identities. For example, after an assignment, the two objects should be equal. Even if you don't define x == y, the two objects should be behaviorally equivalent (see above for the meaning of that phrase) after an assignment.
If you define x == y and x != y, you should maintain the usual identities. For example, these expressions should return something convertible to bool, neither should change its operands, and x == y should have the same result as !(x != y), and vice versa.
If you define inequality operators like x <= y and x < y, you should maintain the usual identities. For example, if x < y and y < z are both true, then x < z should also be true, etc. Similar comments for x >= y and x > y.
If you define inequality operators like x < y and x >= y, you should maintain the usual identities. For example, x < y should have the result as !(x >= y). You can't always do that, but you should get close and you should document any variances. Similar comments for x > y and !(x <= y), etc.
Avoid overloading short-circuiting operators: x || y or x && y. The overloaded versions of these do not short-circuit they evaluate both operands even if the left-hand operand "determines" the outcome, so that confuses users.
Avoid overloading the comma operator: x, y. The overloaded comma operator does not have the same ordering properties that it has when it is not overloaded, and that confuses users.
Don't overload an operator that is non-intuitive to your users. This is called the Doctrine of Least Surprise. For example, altough C++ uses std::cout << x for printing, and although printing is techincally called inserting, and although inserting sort of sounds like what happens when you push an element onto a stack, don't overload myStack << x to push an element onto a stack. It might make sense when you're really tired or otherwise mentally impaired, and a few of your friends might think it's "kewl," but just say No.
Use common sense. If you don't see "your" operator listed here, you can figure it out. Just remember the ultimate goals of operator overloading: to make life easier for your users, in particular to make their code cheaper to write and more obvious.
Caveat: the list is not exhaustive. That means there are other entries that you might consider "missing." I know.

Caveat: the list contains guidelines, not hard and fast rules. That means almost all of the entries have exceptions, and most of those exceptions are not explicitly stated. I know.

Caveat: please don't email me about the additions or exceptions. I've already spent way too much time on this particular answer.

Algorithms

Q: Implement a Algorithm to check if the link list is in Ascending order?
A: template
bool linklist::isAscending() const{
nodeptr ptr = head;
while(ptr->_next)
{
if(ptr->_data > ptr->_next->_data)
return false;

ptr= ptr->_next;
}
return true;
}

Q: Write an algorithm to reverse a link list?
A: template
void linklist::reverselist()
{
nodeptr ptr= head;
nodeptr nextptr= ptr->_next;
while(nextptr)
{
nodeptr temp = nextptr->_next;
nextptr->_next = ptr;
ptr = nextptr;
nextptr = temp;
}
head->_next = 0;
head = ptr;
}

Q: Implement Binary Heap in C++?
A:BinaryHeap.h

```
#ifndef BINARY_HEAP_H_
#define BINARY_HEAP_H_

#include "dsexceptions.h"
#include "vector.h"

// BinaryHeap class
//
// CONSTRUCTION: with an optional capacity (that defaults to 100)
//
// ******************PUBLIC OPERATIONS**********************
// void insert( x ) --> Insert x
// deleteMin( minItem ) --> Remove (and optionally return) smallest item
// Comparable findMin( ) --> Return smallest item
// bool isEmpty( ) --> Return true if empty; else false
```

```
// bool isFull( ) --> Return true if full; else false
// void makeEmpty( ) --> Remove all items
// ******************ERRORS*********************************
// Throws Underflow and Overflow as warranted

template
class BinaryHeap
{
public:
explicit BinaryHeap( int capacity = 100 );

bool isEmpty( ) const;
bool isFull( ) const;
const Comparable & findMin( ) const;

void insert( const Comparable & x );
void deleteMin( );
void deleteMin( Comparable & minItem );
void makeEmpty( );

private:
int currentSize; // Number of elements in heap
vector array; // The heap array

void buildHeap( );
void percolateDown( int hole );
};

#endif
```

BinaryHeap.cpp

```
#include "BinaryHeap.h"

/**
* Construct the binary heap.
* capacity is the capacity of the binary heap.
*/
template
BinaryHeap::BinaryHeap( int capacity )
: array( capacity + 1 ), currentSize( 0 )
{
}

/**
* Insert item x into the priority queue, maintaining heap order.
* Duplicates are allowed.
* Throw Overflow if container is full.
*/
template
void BinaryHeap::insert( const Comparable & x )
{
if( isFull( ) )
throw Overflow( );

// Percolate up
int hole = ++currentSize;
for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
array[ hole ] = array[ hole / 2 ];
array[ hole ] = x;
}

/**
* Find the smallest item in the priority queue.
* Return the smallest item, or throw Underflow if empty.
*/
template
```

```
const Comparable & BinaryHeap::findMin( ) const
{
if( isEmpty( ) )
throw Underflow( );
return array[ 1 ];
}

/**
* Remove the smallest item from the priority queue.
* Throw Underflow if empty.
*/
template
void BinaryHeap::deleteMin( )
{
if( isEmpty( ) )
throw Underflow( );

array[ 1 ] = array[ currentSize-- ];
percolateDown( 1 );
}

/**
* Remove the smallest item from the priority queue
* and place it in minItem. Throw Underflow if empty.
*/
template
void BinaryHeap::deleteMin( Comparable & minItem )
{
if( isEmpty( ) )
throw Underflow( );

minItem = array[ 1 ];
array[ 1 ] = array[ currentSize-- ];
percolateDown( 1 );
}

/**
* Establish heap order property from an arbitrary
* arrangement of items. Runs in linear time.
*/
template
void BinaryHeap::buildHeap( )
{
for( int i = currentSize / 2; i > 0; i-- )
percolateDown( i );
}

/**
* Test if the priority queue is logically empty.
* Return true if empty, false otherwise.
*/
template
bool BinaryHeap::isEmpty( ) const
{
return currentSize == 0;
}

/**
* Test if the priority queue is logically full.
* Return true if full, false otherwise.
*/
template
bool BinaryHeap::isFull( ) const
{
return currentSize == array.size( ) - 1;
}

/**
* Make the priority queue logically empty.
*/
```

```
template
void BinaryHeap::makeEmpty( )
{
currentSize = 0;
}

/**
* Internal method to percolate down in the heap.
* hole is the index at which the percolate begins.
*/
template
void BinaryHeap::percolateDown( int hole )
{
/* 1*/ int child;
/* 2*/ Comparable tmp = array[ hole ];

/* 3*/ for( ; hole * 2 <= currentSize; hole = child )
{
/* 4*/ child = hole * 2;
/* 5*/ if( child != currentSize && array[ child + 1 ] < array[ child ] )
/* 6*/ child++;
/* 7*/ if( array[ child ] < tmp )
/* 8*/ array[ hole ] = array[ child ];
else
/* 9*/ break;
}
/*10*/ array[ hole ] = tmp;
}
```

TestBinaryHeap.cpp

```
#include
#include "BinaryHeap.h"
#include "dsexceptions.h"

// Test program
int main( )
{
int numItems = 10000;
BinaryHeap h( numItems );
int i = 37;
int x;

try
{
for( i = 37; i != 0; i = ( i + 37 ) % numItems )
h.insert( i );
for( i = 1; i < numItems; i++ )
{
h.deleteMin( x );
if( x != i )
cout << "Oops! " << i << endl;
}
for( i = 37; i != 0; i = ( i + 37 ) % numItems )
h.insert( i );
h.insert( 0 );
h.insert( i = 999999 ); // Should overflow
}
catch( Overflow )
{ cout << "Overflow (expected)! " << i << endl; }

return 0;
}
```

Q: Implement Binary Search Tree in C++?

A: BinarySearchTree.h

```
#ifndef BINARY_SEARCH_TREE_H_
#define BINARY_SEARCH_TREE_H_

#include "dsexceptions.h"
#include // For NULL

// Binary node and forward declaration because g++ does
// not understand nested classes.
template
class BinarySearchTree;

template
class BinaryNode
{
Comparable element;
BinaryNode *left;
BinaryNode *right;

BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
: element( theElement ), left( lt ), right( rt ) { }
friend class BinarySearchTree;
};


// BinarySearchTree class
//
// CONSTRUCTION: with ITEM_NOT_FOUND object used to signal failed finds
//
// ******************PUBLIC OPERATIONS**********************
// void insert( x ) --> Insert x
// void remove( x ) --> Remove x
// Comparable find( x ) --> Return item that matches x
// Comparable findMin( ) --> Return smallest item
// Comparable findMax( ) --> Return largest item
// boolean isEmpty( ) --> Return true if empty; else false
// void makeEmpty( ) --> Remove all items
// void printTree( ) --> Print tree in sorted order

template
class BinarySearchTree
{
public:
explicit BinarySearchTree( const Comparable & notFound );
BinarySearchTree( const BinarySearchTree & rhs );
~BinarySearchTree( );

const Comparable & findMin( ) const;
const Comparable & findMax( ) const;
const Comparable & find( const Comparable & x ) const;
bool isEmpty( ) const;
void printTree( ) const;

void makeEmpty( );
void insert( const Comparable & x );
void remove( const Comparable & x );

const BinarySearchTree & operator=( const BinarySearchTree & rhs );

private:
BinaryNode *root;
const Comparable ITEM_NOT_FOUND;

const Comparable & elementAt( BinaryNode *t ) const;

void insert( const Comparable & x, BinaryNode * & t ) const;
```

```
void remove( const Comparable & x, BinaryNode * & t ) const;
BinaryNode * findMin( BinaryNode *t ) const;
BinaryNode * findMax( BinaryNode *t ) const;
BinaryNode * find( const Comparable & x, BinaryNode *t ) const;
void makeEmpty( BinaryNode * & t ) const;
void printTree( BinaryNode *t ) const;
BinaryNode * clone( BinaryNode *t ) const;
};

#endif
```

BinarySearchTree.cpp

```
#include "BinarySearchTree.h"
#include

/**
* Implements an unbalanced binary search tree.
* Note that all "matching" is based on the < method.
*/

/**
* Construct the tree.
*/
template
BinarySearchTree::BinarySearchTree( const Comparable & notFound ) :
root( NULL ), ITEM_NOT_FOUND( notFound )
{
}


/**
* Copy constructor.
*/
template
BinarySearchTree::
BinarySearchTree( const BinarySearchTree & rhs ) :
root( NULL ), ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND )
{
*this = rhs;
}

/**
* Destructor for the tree.
*/
template
BinarySearchTree::~BinarySearchTree( )
{
makeEmpty( );
}

/**
* Insert x into the tree; duplicates are ignored.
*/
template
void BinarySearchTree::insert( const Comparable & x )
{
insert( x, root );
}

/**
* Remove x from the tree. Nothing is done if x is not found.
*/
template
void BinarySearchTree::remove( const Comparable & x )
{
```

```
remove( x, root );
}


/**
* Find the smallest item in the tree.
* Return smallest item or ITEM_NOT_FOUND if empty.
*/
template
const Comparable & BinarySearchTree::findMin( ) const
{
return elementAt( findMin( root ) );
}

/**
* Find the largest item in the tree.
* Return the largest item of ITEM_NOT_FOUND if empty.
*/
template
const Comparable & BinarySearchTree::findMax( ) const
{
return elementAt( findMax( root ) );
}

/**
* Find item x in the tree.
* Return the matching item or ITEM_NOT_FOUND if not found.
*/
template
const Comparable & BinarySearchTree::
find( const Comparable & x ) const
{
return elementAt( find( x, root ) );
}

/**
* Make the tree logically empty.
*/
template
void BinarySearchTree::makeEmpty( )
{
makeEmpty( root );
}

/**
* Test if the tree is logically empty.
* Return true if empty, false otherwise.
*/
template
bool BinarySearchTree::isEmpty( ) const
{
return root == NULL;
}

/**
* Print the tree contents in sorted order.
*/
template
void BinarySearchTree::printTree( ) const
{
if( isEmpty( ) )
cout << "Empty tree" << endl;
else
printTree( root );
}

/**
* Deep copy.
*/
template
```

```
const BinarySearchTree &
BinarySearchTree::
operator=( const BinarySearchTree & rhs )
{
if( this != &rhs )
{
makeEmpty( );
root = clone( rhs.root );
}
return *this;
}

/**
* Internal method to get element field in node t.
* Return the element field or ITEM_NOT_FOUND if t is NULL.
*/
template
const Comparable & BinarySearchTree::
elementAt( BinaryNode *t ) const
{
if( t == NULL )
return ITEM_NOT_FOUND;
else
return t->element;
}

/**
* Internal method to insert into a subtree.
* x is the item to insert.
* t is the node that roots the tree.
* Set the new root.
*/
template
void BinarySearchTree::
insert( const Comparable & x, BinaryNode * & t ) const
{
if( t == NULL )
t = new BinaryNode( x, NULL, NULL );
else if( x < t->element )
insert( x, t->left );
else if( t->element < x )
insert( x, t->right );
else
; // Duplicate; do nothing
}

/**
* Internal method to remove from a subtree.
* x is the item to remove.
* t is the node that roots the tree.
* Set the new root.
*/
template
void BinarySearchTree::
remove( const Comparable & x, BinaryNode * & t ) const
{
if( t == NULL )
return; // Item not found; do nothing
if( x < t->element )
remove( x, t->left );
else if( t->element < x )
remove( x, t->right );
else if( t->left != NULL && t->right != NULL ) // Two children
{
t->element = findMin( t->right )->element;
remove( t->element, t->right );
}
else
{
BinaryNode *oldNode = t;
```

```
t = ( t->left != NULL ) ? t->left : t->right;
delete oldNode;
}
}

/**
* Internal method to find the smallest item in a subtree t.
* Return node containing the smallest item.
*/
template
BinaryNode *
BinarySearchTree::findMin( BinaryNode *t ) const
{
if( t == NULL )
return NULL;
if( t->left == NULL )
return t;
return findMin( t->left );
}

/**
* Internal method to find the largest item in a subtree t.
* Return node containing the largest item.
*/
template
BinaryNode *
BinarySearchTree::findMax( BinaryNode *t ) const
{
if( t != NULL )
while( t->right != NULL )
t = t->right;
return t;
}

/**
* Internal method to find an item in a subtree.
* x is item to search for.
* t is the node that roots the tree.
* Return node containing the matched item.
*/
template
BinaryNode *
BinarySearchTree::
find( const Comparable & x, BinaryNode *t ) const
{
if( t == NULL )
return NULL;
else if( x < t->element )
return find( x, t->left );
else if( t->element < x )
return find( x, t->right );
else
return t; // Match
}
/****** NONRECURSIVE VERSION*************************
template
BinaryNode *
BinarySearchTree::
find( const Comparable & x, BinaryNode *t ) const
{
while( t != NULL )
if( x < t->element )
t = t->left;
else if( t->element < x )
t = t->right;
else
return t; // Match

return NULL; // No match
}
```

```
**************************************************/

/**
* Internal method to make subtree empty.
*/
template
void BinarySearchTree::
makeEmpty( BinaryNode * & t ) const
{
if( t != NULL )
{
makeEmpty( t->left );
makeEmpty( t->right );
delete t;
}
t = NULL;
}

/**
* Internal method to print a subtree rooted at t in sorted order.
*/
template
void BinarySearchTree::printTree( BinaryNode *t ) const
{
if( t != NULL )
{
printTree( t->left );
cout << t->element << endl;
printTree( t->right );
}
}

/**
* Internal method to clone subtree.
*/
template
BinaryNode *
BinarySearchTree::clone( BinaryNode * t ) const
{
if( t == NULL )
return NULL;
else
return new BinaryNode( t->element, clone( t->left ), clone( t->right ) );
}
```

TestBinarySearchTree.cpp

```
#include
#include "BinarySearchTree.h"

// Test program
int main( )
{
const int ITEM_NOT_FOUND = -9999;
BinarySearchTree t( ITEM_NOT_FOUND );
int NUMS = 4000;
const int GAP = 37;
int i;

cout << "Checking... (no more output means success)" << endl;

for( i = GAP; i != 0; i = ( i + GAP ) % NUMS )
t.insert( i );

for( i = 1; i < NUMS; i+= 2 )
t.remove( i );
```

```
if( NUMS < 40 )
t.printTree( );
if( t.findMin( ) != 2 || t.findMax( ) != NUMS - 2 )
cout << "FindMin or FindMax error!" << endl;

for( i = 2; i < NUMS; i+=2 )
if( t.find( i ) != i )
cout << "Find error1!" << endl;

for( i = 1; i < NUMS; i+=2 )
{
if( t.find( i ) != ITEM_NOT_FOUND )
cout << "Find error2!" << endl;
}

BinarySearchTree t2( ITEM_NOT_FOUND );
t2 = t;

for( i = 2; i < NUMS; i+=2 )
if( t2.find( i ) != i )
cout << "Find error1!" << endl;

for( i = 1; i < NUMS; i+=2 )
{
if( t2.find( i ) != ITEM_NOT_FOUND )
cout << "Find error2!" << endl;
}


return 0;
}
```

Friends

Q: What is a friend?
A: Something to allow your class to grant access to another class or function.

Friends can be either functions or other classes. A class grants access privileges to its friends. Normally a developer has political and technical control over both the friend and member functions of a class (else you may need to get permission from the owner of the other pieces when you want to update your own class).

Q: Do friends violate encapsulation?
A: No! If they're used properly, they enhance encapsulation.

You often need to split a class in half when the two halves will have different numbers of instances or different lifetimes. In these cases, the two halves usually need direct access to each other (the two halves used to be in the same class, so you haven't increased the amount of code that needs direct access to a data structure; you've simply reshuffled the code into two classes instead of one). The safest way to implement this is to make the two halves friends of each other.

If you use friends like just described, you'll keep private things private. People who don't understand this often make naive efforts to avoid using friendship in situations like the above, and often they actually destroy encapsulation. They either use public data (grotesque!), or they make the data accessible between the halves via public get() and set() member functions. Having a public get() and set() member function for a private datum is OK only when the private datum "makes sense" from outside the class (from a user's perspective). In many cases, these get()/set() member functions are almost as bad as public data: they hide (only) the name of the private datum, but they don't hide the existence of the private datum.

Similarly, if you use friend functions as a syntactic variant of a class's public access functions, they don't violate encapsulation any more than a member function violates encapsulation. In other words, a class's friends don't violate the encapsulation barrier: along with the class's member functions, they are the encapsulation barrier.

(Many people think of a friend function as something outside the class. Instead, try thinking of a friend function as part of the class's public interface. A friend function in the class declaration doesn't violate encapsulation any more than a public member function violates encapsulation: both have exactly the same authority with respect to accessing the class's non-public parts.)

Q: What are some advantages/disadvantages of using friend functions?
A: They provide a degree of freedom in the interface design options.

Member functions and friend functions are equally privileged (100% vested). The major difference is that a friend function is called like f(x), while a member function is called like x.f(). Thus the ability to choose between member functions (x.f()) and friend functions (f(x)) allows a designer to select the syntax that is deemed most readable, which lowers maintenance costs.

The major disadvantage of friend functions is that they require an extra line of code when you want dynamic binding. To get the effect of a virtual friend, the friend function should call a hidden (usually protected) virtual member function. This is called the Virtual Friend Function Idiom. For example:

```
class Base {
public:
friend void f(Base& b);
...
protected:
virtual void do_f();
...
};

inline void f(Base& b)
{
b.do_f();
}

class Derived : public Base {
public:
...
protected:
virtual void do_f(); // "Override" the behavior of f(Base& b)
...
};

void userCode(Base& b)
{
f(b);
}
```

The statement f(b) in userCode(Base&) will invoke b.do_f(), which is virtual. This means that Derived::do_f() will get control if b is actually a object of class Derived. Note that Derived overrides the behavior of the protected virtual member function do_f(); it does not have its own variation of the friend function, f(Base&).

Q: What does it mean that "friendship isn't inherited, transitive, or reciprocal"?
A: Just because I grant you friendship access to me doesn't automatically grant your kids access to me, doesn't automatically grant your friends access to me, and doesn't automatically grant me access to you.

I don't necessarily trust the kids of my friends. The privileges of friendship aren't inherited. Derived classes of a friend aren't necessarily friends. If class Fred declares that class Base is a friend, classes derived from Base don't have any automatic special access rights to Fred objects. I don't necessarily trust the friends of my friends. The privileges of friendship aren't transitive. A

friend of a friend isn't necessarily a friend. If class Fred declares class Wilma as a friend, and class Wilma declares class Betty as a friend, class Betty doesn't necessarily have any special access rights to Fred objects.
You don't necessarily trust me simply because I declare you my friend. The privileges of friendship aren't reciprocal. If class Fred declares that class Wilma is a friend, Wilma objects have special access to Fred objects but Fred objects do not automatically have special access to Wilma objects.

Q: Should my class declare a member function or a friend function?
A: Use a member when you can, and a friend when you have to.

Sometimes friends are syntactically better (e.g., in class Fred, friend functions allow the Fred parameter to be second, while members require it to be first). Another good use of friend functions are the binary infix arithmetic operators. E.g., aComplex + aComplex should be defined as a friend rather than a member if you want to allow aFloat + aComplex as well (member functions don't allow promotion of the left hand argument, since that would change the class of the object that is the recipient of the member function invocation).

In other cases, choose a member function over a friend function.

Q: How can I provide printing for my class Fred?
A: Use operator overloading to provide a friend left-shift operator, operator<<.

```
#include

class Fred {
public:
friend std::ostream& operator<< (std::ostream& o, const Fred& fred);
...
private:
int i_; // Just for illustration
};

std::ostream& operator<< (std::ostream& o, const Fred& fred)
{
return o << fred.i_;
}

int main()
{
Fred f;
std::cout << "My Fred object: " << f << "\n";
...
}
```

We use a non-member function (a friend in this case) since the Fred object is the right-hand operand of the << operator. If the Fred object was supposed to be on the left hand side of the <<

(that is, myFred << std::cout rather than std::cout << myFred), we could have used a member function named operator<<.

Note that operator<< returns the stream. This is so the output operations can be cascaded.

Q: But shouldn't I always use a printOn() method rather than a friend function?
A: No.

The usual reason people want to always use a printOn() method rather than a friend function is because they wrongly believe that friends violate encapsulation and/or that friends are evil. These beliefs are naive and wrong: when used properly, friends can actually enhance encapsulation.

This is not to say that the printOn() method approach is never useful. For example, it is useful when providing printing for an entire hierarchy of classes. But if you use a printOn() method, it should normally be protected, not public.

For completeness, here is "the printOn() method approach." The idea is to have a member function, often called printOn(), that does the actual printing, then have operator<< call that printOn() method. When it is done wrongly, the printOn() method is public so operator<< doesn't have to be a friend it can be a simple top-level function that is neither a friend nor a member of the class. Here's some sample code:

```
#include

class Fred {
public:
void printOn(std::ostream& o) const;
...
};

// operator<< can be declared as a non-friend [NOT recommended!]
std::ostream& operator<< (std::ostream& o, const Fred& fred);

// The actual printing is done inside the printOn() method [NOT recommended!]
void Fred::printOn(std::ostream& o) const
{
...
}

// operator<< calls printOn() [NOT recommended!]
std::ostream& operator<< (std::ostream& o, const Fred& fred)
{
fred.printOn(o);
return o;
```

}
People wrongly assume that this reduces maintenance cost "since it avoids having a friend function." This is a wrong assumption because:

The member-called-by-top-level-function approach has zero benefit in terms of maintenance cost. Let's say it takes N lines of code to do the actual printing. In the case of a friend function, those N lines of code will have direct access to the class's private/protected parts, which means whenever someone changes the class's private/protected parts, those N lines of code will need to be scanned and possibly modified, which increases the maintenance cost. However using the printOn() method doesn't change this at all: we still have N lines of code that have direct access to the class's private/protected parts. Thus moving the code from a friend function into a member function does not reduce the maintenance cost at all. Zero reduction. No benefit in maintenance cost. (If anything it's a bit worse with the printOn() method since you now have more lines of code to maintain since you have an extra function that you didn't have before.)
The member-called-by-top-level-function approach makes the class harder to use, particularly by programmers who are not also class designers. The approach exposes a public method that programmers are not supposed to call. When a programmer reads the public methods of the class, they'll see two ways to do the same thing. The documentation would need to say something like, "This does exactly the same as that, but don't use this; instead use that." And the average programmer will say, "Huh? Why make the method public if I'm not supposed to use it?" In reality the only reason the printOn() method is public is to avoid granting friendship status to operator<<, and that is a notion that is somewhere between subtle and incomprehensible to a programmer who simply wants to use the class.
Net: the member-called-by-top-level-function approach has a cost but no benefit. Therefore it is, in general, a bad idea.

Note: if the printOn() method is protected or private, the second objection doesn't apply. There are cases when that approach is reasonable, such as when providing printing for an entire hierarchy of classes. Note also that when the printOn() method is non-public, operator<< needs to be a friend.

Q: How can I provide input for my class Fred?
A: Use operator overloading to provide a friend right-shift operator, operator>>. This is similar to the output operator, except the parameter doesn't have a const: "Fred&" rather than "const Fred&".

```
#include

class Fred {
public:
friend std::istream& operator>> (std::istream& i, Fred& fred);
...
private:
int i_; // Just for illustration
};
```

```
std::istream& operator>> (std::istream& i, Fred& fred)
{
return i >> fred.i_;
}

int main()
{
Fred f;
std::cout << "Enter a Fred object: ";
std::cin >> f;
...
}
```
Note that operator>> returns the stream. This is so the input operations can be cascaded and/or used in a while loop or if statement.

Q: How can I provide printing for an entire hierarchy of classes?
A: Provide a friend operator<< that calls a protected virtual function:

```
class Base {
public:
friend std::ostream& operator<< (std::ostream& o, const Base& b);
...
protected:
virtual void printOn(std::ostream& o) const;
};

inline std::ostream& operator<< (std::ostream& o, const Base& b)
{
b.printOn(o);
return o;
}

class Derived : public Base {
protected:
virtual void printOn(std::ostream& o) const;
};
```
The end result is that operator<< acts as if it were dynamically bound, even though it's a friend function. This is called the Virtual Friend Function Idiom.

Note that derived classes override printOn(std::ostream&) const. In particular, they do not provide their own operator<<.

Naturally if Base is an ABC, Base::printOn(std::ostream&) const can be declared pure virtual using the "= 0" syntax.

Other

Q: Should I use NULL or 0?
A: In C++, the definition of NULL is 0, so there is only an aesthetic difference. I prefer to avoid macros, so I use 0. Another problem with NULL is that people sometimes mistakenly believe that it is different from 0 and/or not an integer. In pre-standard code, NULL was/is sometimes defined to something unsuitable and therefore had/has to be avoided. That's less common these days.

Q: Can inline functions have a recursion?
A: No. Syntax wise it is allowed. But then the function is no longer Inline. As the compiler will never know how deep the recursion is at compilation time.

Q: Explain the scope resolution operator?
A: It permits a program to reference an identifier in the global scope that has been hidden by another identifier with the same name in the local scope.

Q: Write a function to display an integer in a binary format.
A: void displayBits(unsigned value)
{
const int SHIFT = 8 * sizeof(unsigned) - 1;
const unsigned MASK = 1<< SHIFT;
cout << setw(10) << value << " = ";
for (unsigned i = 1; i <= SHIFT + 1; i++)
{
cout << (value & MASK ? '1' : '0');
value <<= 1;

if (i % 8 == 0) // output a space after bits
cout << ' ';
}
cout << endl;
}

You can do the same using divide by 2, until the number is greator than 0. But you will have to use stack to print it in reverse order.

Q: How many ways are there to initialize an int with a constant?
A: (a) int foo = 123;
(b) int bar(123);

Q: What is your reaction to this line of code?
delete this;
A: It is not a good programming Practice. A good programmer will insist that you should absolutely never use the statement if the class is to be used by other programmers an instantiated

as static, extern, or automatic objects. That much should be obvious. The code has two built-in pitfalls. First, if it executes in a member function for an extern, static, or automatic object, the program will probably crash as soon as the delete statement executes. There is no portable way for an object to tell that it was instantiated on the heap, so the class cannot assert that its object is properly instantiated. Second, when an object commits suicide this way, the using program might not know about its demise. As far as the instantiating program is concerned, the object remains in scope and continues to exist even though the object did itself in. Subsequent dereferencing of the caller can and usually does lead to disaster. I think that the language rules should disallow the idiom, but that's another matter.

Q: What are the debugging methods you use when came across a problem?
A: Debugging with tools like :
(a) GDB, DBG, Forte, Visual Studio.
(b) Analyzing the Core dump.
(c) Using tusc to trace the last system call before crash.
(d) Putting Debug statements in the program source code.

Q: How the compiler arranges the various sections in the executable image?
A: The executable had following sections:
(a) Data Section (uninitialized data variable section, initialized data variable section)
(b) Code Section
(c) Remember that all static variables are allocated in the initialized variable section.

Q: Can you think of a situation where your program would crash without reaching the breakball, which you set at the beginning of main()?
A: C++ allows for dynamic initialization of global variables before main() is invoked. It is possible that initialization of global will invoke some function. If this function crashes the rash will occur before main() is entered.

Q: Why do C++ compilers need name mangling?
A: Name mangling is the rule according to which C++ changes function's name into function signature before passing that function to a linker. This is how the linker differentiates between different functions with the same name.

Q: What is difference between template and macro?
A: In C++ there is a major difference between a template and a macro. A
macro is merely a string that the compiler replaces with the value that was defined.
E.g. #define STRING_TO_BE_REPLACED "ValueToReplaceWith"

A template is a way to make functions independent of data-types. This cannot be accomplished using macros.
E.g. a sorting function doesn't have to care whether it's sorting integers or letters since the same algorithm might apply anyway.

Q: What are C++ storage classes?

A:
auto: the default. Variables are automatically created and initialized when they are defined and are destroyed at the end of the block containing their definition. They are not visible outside that block.
register: a type of auto variable. a suggestion to the compiler to use a CPU register for performance.
static: a variable that is known only in the function that contains its definition but is never destroyed and retains its value between calls to that function. It exists from the time the program begins execution.
extern: a static variable whose definition and placement is determined when all object and library modules are combined (linked) to form the executable code file. It can be visible outside the file where it is defined.

Q: What are storage qualifiers in C++ ?
A: They are:
Const: Indicates that memory once initialized, should not be altered by a program.
Volatile: Indicates that the value in the memory location can be altered even though nothing in the program code modifies the contents. for example if you have a pointer to hardware location that contains the time, where hardware changes the value of this pointer variable and not the program. The intent of this keyword to improve the optimization ability of the compiler.
Mutable: Iindicates that particular member of a structure or class can be altered even if a particular structure variable, class, or class member function is constant.

```
struct data
{
char name[80];
mutable double salary;
}
const data MyStruct = { "Satish Shetty", 1000 };
//initlized by complier
strcpy ( MyStruct.name, "Shilpa Shetty"); // compiler
error
MyStruct.salaray = 2000 ; // complier is happy
allowed
```

Q: What is reference ??
A: reference is a name that acts as an alias, or alternative name, for a previously defined variable or an object. prepending variable with "&" symbol makes it as reference.
for example:

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

Q: What is passing by reference?
A: Method of passing arguments to a function which takes parameter of type reference.
for example:

```
void swap( int & x, int &amp;amp;amp;amp; y )
```

```
{
int temp = x;
x = y;
y = temp;
}

int a=2, b=3;
swap( a, b );
```
Basically, inside the function there won't be any copy of the arguments "x" and "y" instead they refer to original variables a and b. so no extra memory needed to pass arguments and it is more efficient.

Q: When do use "const" reference arguments in function?
A:
a) Using const protects you against programming errors that inadvertently alter data.
b) Using const allows function to process both const and non-const actual arguments, while a function without const in the prototype can only accept non constant arguments.
c) Using a const reference allows the function to generate and use a temporary variable appropriately.

Q: When are temporary variables created by C++ compiler?
A: Provided that function parameter is a "const reference", compiler generates temporary variable
in following2 ways.
a) The actual argument is the correct type, but it
isn't Lvalue
```
double Cube(const double & num)
{
num = num * num * num;
return num;
}
double temp = 2.0;
double value = cube(3.0 + temp); // argument is a expression and not a Lvalue;
```

b) The actual argument is of the wrong type, but of a type that can be converted to the correct type
```
long temp = 3L;
double value = cuberoot ( temp); // long to double conversion
```

Q: What problem does the namespace feature solve?
A: Multiple providers of libraries might use common global identifiers causing a name collision when an application tries to link with two or more such libraries. The namespace feature surrounds a library's external declarations with a unique namespace that eliminates the potential for those collisions.
```
namespace [identifier] { namespace-body }
```

A namespace declaration identifies and assigns a name to a declarative region.
The identifier in a namespace declaration must be unique in the declarative region in which it is used. The identifier is the name of the namespace and is used to reference its members.

Q: What is the use of 'using' declaration?
A: A using declaration makes it possible to use a name from a namespace without the scope operator.

Q: What is an Iterator class?
A: A class that is used to traverse through the objects maintained by a container class. There are _ve 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. Something like a pointer.

Q: What do you mean by Stack unwinding?
A: It is a process during exception handling when the destructor is called for all local objects in the stack between the place where the exception was thrown and where it is caught.

Q: Implementation of string length using pointer hopping
A: #include

```
// Test to see if pointer hopping is worthwhile.
// strlen implemented with usual indexing mechanism.
int strlen1( const char str[ ] )
{
int i;

for( i = 0; str[ i ] != '\0'; i++ )
;

return i;
}

// strlen implemented with pointer hopping.
int strlen2( const char *str )
{
const char *sp = str;

while( *sp++ != '\0' )
```

```
;

return sp - str - 1;
}

// Quick and dirty main
int main( )
{
char str[ 512 ];

cout << "Enter strings; use EOF-marker to terminate: " << endl;
while( cin >> str )
{
if( strlen1( str ) != strlen2( str ) )
cerr << "Oops!!!" << endl;
}

return 0;
}
```

Q: What is inline function?
A: The inline keyword tells the compiler to substitute the code within the function de_nition for every instance of a function call. However, substitution occurs only at the compiler's discretion. For example, the compiler does not inline a function if its address is taken or if it is too large to inline.

Q: What is name mangling in C++??
A: The process of encoding the parameter types with the function/method name into a unique name is called name mangling. The inverse process is called demangling.
For example Foo::bar(int, long) const is mangled as 'bar__C3Fooil'.
For a constructor, the method name is left out.
That is Foo::Foo(int, long) const is mangled as ‘C3Fooil’

Q: Can you think of a situation where your program would crash without reaching the breakpoint which you set at the beginning of main()?
A: C++ allows for dynamic initialization of global variables before main() is invoked. It is possible that initialization of global will invoke some function. If this function crashes the crash will occur before main() is entered. Name two cases where you MUST use initialization list as opposed to assignment in constructors. Both non-static const data members and reference data members cannot be assigned values; instead, you should use initialization list to initialize them.

Q: What does it mean to declare a...
(a) member function as virtual?
(b) member variable as static?
(c) function as static?

(d) destructor as static?

A:

(a) C++ virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. The whole function body can be replaced with a new set of implementation in the derived class. The concept of c++ virtual functions is different from C++ Function overloading. The difference between a non-virtual c++ member function and a virtual member function is, the non-virtual member functions are resolved at compile time. This mechanism is called static binding. Where as the c++ virtual member functions are resolved during run-time. This mechanism is known as dynamic binding.

(b) The static keyword allows a variable to maintain its value among different function calls. If the value of a static variable changes when the variable has been accessed, the variable keeps the new value. If the same variable gets accessed again, it would be holding its most recent value. This is possible because, when the static variable is declared, the compiler uses a separate memory area to store it. By doing this, when the value of the static variable gets changed, it is updated in the memory it is occupying. And because this memory is separate, the compiler can monitor its values even when its function exits.

(c) A static member function can access only static member data, static member functions and data and functions outside the class. A static member function can be called, even when a class is not instantiated. A static member function cannot be declared virtual. A static member function cannot have access to the 'this' pointer of the class.

(d) a "static destructor" is a static member function of the class that accepts one argument - a pointer to the object of that class to be destroyed. It is probably used along with "a factory method", when there is a need to restrict the creation of instances of some class to free store only and/or perform additional steps before or after creation of an object. Similar steps may need to be taken before and/or after destroying an instance.

Q: What is faster ++i or i++, where i is an interger variable?
A: The answer to this lies in the fact, how they used. With ++i(PreIncrement) the variable is incremented and new value is returned. So it requires one instruction to increment the variable.

In case of i++(post Increment), the old value has to be returned or used in the expression and then the variable is incrememented after the expression is evaluated. Since you need one instruction to save the old value to be used in the expression and other instruction to increment the variable, its comparatively slower.

Q: Find the size of an interger data type with out using sizeof() function?
A: #include
int main()
{
int *i ;
int *j = i + 1;
cout << " size of an integer variable i = " << (int)j - (int)i << endl;
}

Q: How can I make it so keys pressed by users are not echoed on the screen?

A: This is not a standard C++ feature. C++ doesn't even require your system to have a keyboard or a screen. That means every operating system and vendor does it somewhat differently.

Please read the documentation that came with your compiler for details on your particular installation.

Q: Why can't I open a file in a different directory such as "..\test.dat"?
A: Because " " is a tab character.

You should use forward slashes in your filenames, even on operating systems that use backslashes (DOS, Windows, OS/2, etc.). For example:

```
#include
#include

int main()
{
#if 1
std::ifstream file("../test.dat"); // RIGHT!
#else
std::ifstream file(".. est.dat"); // WRONG!
#endif

...
}
```

Remember, the backslash ("\") is used in string literals to create special characters: "\n" is a newline, "\b" is a backspace, and " " is a tab, "\a" is an "alert", "\v" is a vertical-tab, etc. Therefore the file name "\version\next\alpha\beta est.dat" is interpreted as a bunch of very funny characters. To be safe, use "/version/next/alpha/beta/test.dat" instead, even on systems that use a "\" as the directory separator. This is because the library routines on these operating systems handle "/" and "\" interchangeably.

Of course you could use "\\version\\next\\alpha\\beta\ est.dat", but that might hurt you (there's a non-zero chance you'll forget one of the "\"s, a rather subtle bug since most people don't notice it) and it can't help you (there's no benefit for using "\\" over "/"). Besides "/" is more portable since it works on all flavors of Unix, Plan 9, Inferno, all Windows, OS/2, etc., but "\\" works only on a subset of that list. So "\\" costs you something and gains you nothing: use "/" instead.

Q: How can I open a stream in binary mode?
A: Use std::ios::binary.

Some operating systems differentiate between text and binary modes. In text mode, end-of-line sequences and possibly other things are translated; in binary mode, they are not. For example, in text mode under Windows, "\r\n" is translated into "\n" on input, and the reverse on output.

To read a file in binary mode, use something like this:

```
#include
#include
#include

void readBinaryFile(const std::string& filename)
{
std::ifstream input(filename.c_str(), std::ios::in | std::ios::binary);
char c;
while (input.get(c)) {
...do something with c here...
}
}
```

Note: input >> c discards leading whitespace, so you won't normally use that when reading binary files.

Q: How can I "reopen" std::cin and std::cout in binary mode?
A: This is implementation dependent. Check with your compiler's documentation.

For example, suppose you want to do binary I/O using std::cin and std::cout.

Unfortunately there is no standard way to cause std::cin, std::cout, and/or std::cerr to be opened in binary mode. Closing the streams and attempting to reopen them in binary mode might have unexpected or undesirable results.

On systems where it makes a difference, the implementation might provide a way to make them binary streams, but you would have to check the implementation specifics to find out.