RightOn

Try **RightOn**, the ultimate companion for your mouse!

RightOn is a Windows 3.1 utility which allows you to turn your right and/or middle buttons of your mouse into a left-mouse double click or virtually any keystroke that you could type from the keyboard.

RightOn also gives you the ability to "push" foreground windows to the background by simply clicking the right mouse button in the caption area of the foreground window.

You might turn your right mouse button into a left-mouse double-click and launch applications at the touch of a button. Or maybe you want to set up the middle button to send the Return key, to make dialog boxes which appear on the screen go away.

Throw in the ability to modify the simulated keystrokes with any combination of the Shift, Alt and Control keys, and you've got a configurable, personalized power tool at your fingertips!

RightOn is available by anonymous ftp from **`ftp.cica.indiana.edu`**, in the /pub/pc/win3/desktop directory as **RITEON21.ZIP**. It is also available on CompuServe in the Windows Shareware forum (GO WINSHARE), America OnLine, and thousands of bulletin board systems all over the world.

C++ Frequently Asked Questions

Introduction

Document:	Frequently-Asked-Questions for **comp.lang.c++**
Revision:	Sep 13, 1993
Author:	**Marshall P. Cline, Ph.D**. Paradigm Shift, Inc. One Park St. / Norwood, NY 13668 voice: 315-353-6100 fax: 315-353-6110 email: cline@parashift.com
WinHelp by:	**Steven J. McCarthy** voice 206-932-3351 email: smccrew@hebron.connected.com
Copyright:	
WARRANTY:	
Availability:	This is available via anonymous ftp from: sun.soe.clarkson.edu [128.153.12.3] in the file: pub/C++/FAQ

Without FTP:	You can also get it by sending electronic mail:
	`To: archive-server@sun.soe.clarkson.edu`
	`Subject: send C++/FAQ`
	This will help those who don't have ftp. (note: I hear the mail server is down; if you have problems, send details and I'll look into it).

See also:	comp.lang.c's FAQ appears at the beginning of every month in that newsgroup, and is maintained by Steve Summit (scs@adm.mit.edu).

Table of Contents

SECTION 1A:Table of Contents

SECTION 1B: Nomenclature and Common Abbreviations

TERM	MEANING
ctor	constructor
copy-ctor	copy constructor (also `X(const X&)', pronounced `X-X-ref')
dtor	destructor
fn	function
fns	functions
ptr	pointer, a C/C++ construct declared by: int * p;
ref	reference, a C++ construct declared by: int & r;
OO	object-oriented
OOP	object-oriented programming
OOPL	object-oriented programming language
method	an alternate term for `member function'

SECTION 2: Environmental/managerial issues

Q1: What is C++? What is OOP?

Q2: What are some advantages of C++?

Q3: Who uses C++?

Q4: Are there any C++ standardization efforts underway?

Q5: Where can I ftp a copy of the latest ANSI-C++ draft standard?

Q6: Is C++ backward compatible with ANSI-C?

Q7: How long does it take to learn C++?

SECTION 3: Basics of the paradigm

Q8: What is a class?

Q9: What is an object?

Q10: What is a reference?

Q11: What happens if you assign to a reference?

Q12: How can you reseat a reference to make it refer to a different object?

Q13: When should I use references, and when should I use pointers?

Q14: What are inline fns? What are their advantages? How are they declared?

SECTION 4: Constructors and destructors

Q15: What is a constructor? Why would I ever use one?

Q16: How can I make a constructor call another constructor as a primitive?

Q17: What are destructors really for? Why would I ever use them?

SECTION 5: Operator overloading

SECTION 6: Friends

Q21: What is a `friend'?

Q22: Do `friends' violate encapsulation?

Q23: What are some advantages/disadvantages of using friends?

Q24: What does it mean that `friendship is neither inherited nor transitive'?

Q25: When would I use a member function as opposed to a friend function?

SECTION 7: Input/output via <iostream.h> and <stdio.h>

SECTION 8: Freestore management

SECTION 9: Debugging and error handling

SECTION 10: Const correctness

SECTION 11: Inheritance

SUBSECTION 11A: Inheritance -- virtual functions

SUBSECTION 11B: Inheritance -- conformance

Q56: Can I `revoke' or `hide' public member fns inherited from my base class?

Q57: Is a `Circle' a kind-of an `Ellipse'?

Q58: Are there other options to the `Circle is/isnot kind-of Ellipse' dilemma?

SUBSECTION 11C: Inheritance -- access rules

SUBSECTION 11D: Inheritance -- constructors and destructors

Q62: Why does base ctor get *base*'s virtual fn instead of the derived version?

Q63: Does a derived class dtor need to explicitly call the base destructor?

SUBSECTION 11E: Inheritance -- private and protected inheritance

SECTION 12: Abstraction

SECTION 13: Style guidelines

SECTION 14: C++/Smalltalk differences and keys to learning C++

SECTION 15: Reference and value semantics

SECTION 16: Linkage-to/relationship-with C

SECTION 17: Pointers to member functions

SECTION 18: Container classes and templates

SECTION 19: Nuances of particular implementations

SECTION 20: Miscellaneous technical and environmental issues

SUBSECTION 20A: Miscellaneous technical issues:

SUBSECTION 20B: Miscellaneous environmental issues:

Index of All Questions

SECTION 2: Environmental/managerial issues

SECTION 3: Basics of the paradigm

SECTION 4: Constructors and destructors

SECTION 5: Operator overloading

SECTION 6: Friends

SECTION 7: Input/output via <iostream.h> and <stdio.h>

SECTION 8: Freestore management

SECTION 9: Debugging and error handling

SECTION 10: Const correctness

SECTION 11: Inheritance

SUBSECTION 11A: Inheritance -- virtual functions

SUBSECTION 11B: Inheritance -- conformance

SUBSECTION 11C: Inheritance -- access rules

SUBSECTION 11D: Inheritance -- constructors and destructors

SUBSECTION 11E: Inheritance -- private and protected inheritance

SECTION 12: Abstraction

SECTION 13: Style guidelines

SECTION 14: C++/Smalltalk differences and keys to learning C++

SECTION 15: Reference and value semantics

SECTION 16: Linkage-to/relationship-with C

SECTION 17: Pointers to member functions

SECTION 18: Container classes and templates

SECTION 19: Nuances of particular implementations

SECTION 20: Miscellaneous technical and environmental issues

SUBSECTION 20A: Miscellaneous technical issues:

SUBSECTION 20B: Miscellaneous environmental issues:

Q131: Is there a TeX or LaTeX macro that fixes the spacing on `C++'?

Q132: Where can I access C++2LaTeX, a LaTeX pretty printer for C++ source?

Q133: Where can I access `tgrind', a pretty printer for C++/C/etc source?

Q134: Is there a C++-mode for GNU emacs? If so, where can I get it?

Q135: What is `InterViews'?

Q136: Where can I get OS-specific questions answered (ex: BC++, DOS, Windows, etc)?

Q137: Why does my DOS C++ program says `Sorry: floating point code not linked'?

Q1:What is C++? What is OOP?

C++ can be used simply as `a better C', but that is not its real advantage. C++ is an object-oriented programming language (OOPL). OOPLs appear to be the current `top shelf' in the development of programming languages that can manage the complexity of large software systems.

Some OOP hype: software engineering is `failing' to provide the current users demands for large, complex software systems. But this `failure' is actually due to SE's *successes*. In other words, structured programming was developed to allow software engineers to design/build HUGE software systems (that's success). When users saw how successful these systems were, they said, `More give me MOOORRRREEEE'. They wanted more power, more features, more flexibility. 100K line systems are almost commonplace nowadays, and they still want more. Structured programming techniques, some say, begin to break down around 100K lines (the complexity gives the design team too many headaches, and fixing one problem breaks 5 more, etc). So pragmatics demands a better paradigm than structured programming. Hence OO-design.

Q2: What are some advantages of C++?

GROWTH OF C++: C++ is by far the most popular OOPL. Knowing C++ is a good resume-stuffer. But don't just use it as a better C, or you won't be using all its power. Like any quality tool, C++ must be used the way it was designed to be used. The number of C++ users is doubling every 7.5 to 9 months. This exponential growth can't continue forever(!), but it is becoming a significant chunk of the programming market (it's already the dominant OOPL).

ENCAPSULATION: For those of you who aren't on a team constructing software mega-systems, what does C++ buy you? Here's a trivial example. Suppose you want a `Foible' data type. One style of doing this in `C' is to create a `Foible.h' file that holds the `public interface', then stick all the implementation into a `Foible.c' file. Encapsulation (hiding the details) can be achieved by making all data elements in `Foible.c' be `static'. But that means you only get one `Foible' in the entire system, which is ok if `Foible' is a Screen or perhaps a HardDisk, but is lousy if Foible is a complex number or a line on the screen, etc. Read on to see how it's done in `C' vs `C++'.

MULTIPLE INSTANCES: The `C' solution to the above `multiple instances' problem is to wrap all the data members in a struct (like a Pascal `record'), then pass these structs around as if they were the `ComplexNumber' or whatever. But this loses encapsulation. Other techniques can be devised which allow both multiple instances and encapsulation, however these lose on other accounts (ex: typedef'ing `Foible' to be `void*' loses type safety, and wrapping a `void*' in the Foible struct loses an extra layer of indirection). So the `module' technique loses multiple instantiations, but the `struct' technique loses encapsulation. C++ allows you to combine the best of both worlds - you can have what amount to structs whose data is hidden.

INLINE FUNCTION CALLS: The `encapsulated C' solution above requires a function call to access even trivial fields of the data type (if you allowed direct access to the struct's fields, the underlying data structure would become virtually impossible to change since too many pieces of code would *rely* on it being the `old' way). Function call overhead is small, but can add up. C++ provides a solution by allowing function calls to be expanded `inline', so you have: the (1) safety of encapsulation, (2) convenience of multiple instances, (3) speed of direct access. Furthermore the parameter types of these inlinefunctions are checked by the compiler, an improvement over C's #define macros.

OVERLOADING OPERATORS: For the `ComplexNumber' example, you want to be able to use it in an expression `just as if' it was a builtin type like int or float.C++ allows you to overload operators, so you can tell the compiler what it means for two complex numbers to be added, subtracted, multiplied, etc. This gives you: `z0 = (z1 + z2) * z3 / z4`; Furthermore you might want string1+string2 to mean string concatenation, etc. One of the goals of C++ is to make user defined types `look like' builtin types. You can even have `smart pointers', which means a pointer `p' could actually be a user defined data type that `points' to a disk record (for example). `Dereferencing' such a pointer (ex: i=*p;) means ``seek to the location on disk where p `points' and return its value''. Also statements like p->field=27; can store things on disk, etc. If later on you find you can fit the entire pointed-to data structure in memory, you just change the user-defined pseudo-pointer type and recompile. All the code that used these `pseudo pointers' doesn't need to be changed at all.

INHERITANCE: We still have just scratched the surface. In fact, we haven't even gotten to the `object-oriented' part yet! Suppose you have a Stack data type with operations push, pop, etc. Suppose you want an InvertableStack, which is `just like' Stack except it also has an `invert' operation. In `C' style, you'd have to either (1) modify the existing Stack module (trouble if `Stack' is being used by others), or (2) copy Stack into another file and text edit that file (results in lots of code duplication, another chance to break something tricky in the Stack part of InvertableStack, and especially twice as much code to maintain). C++ provides a much cleaner solution: inheritance. You say `InvertableStack inherits everything from Stack, and InvertableStack adds the invert operation'. Done. Stack itself remains `closed' (untouched, unmodified), and InvertableStack doesn't duplicate the code for push/pop/etc.

POLYMORPHISM: The real power of OOP isn't just inheritance, but is the ability to pass an InvertableStack around as if it actually were a Stack. This is `safe' since (in C++ at least) the is-a relation follows public inheritance (ie: a InvertableStack is-a Stack that can also invert itself). Polymorphism is easiest to understand from an example, so here's a `classic': a graphical draw package might deal with Circles, Squares, Rectangles, general Polygons, and Lines. All of these are Shapes. Most of the draw package's functions need a `Shape'

parameter (as opposed to some particular kind of shape like Square). Ex: if a Shape is picked by a mouse, the Shape might get dragged across the screen and placed into a new location. Polymorphism allows the code to work correctly even if the compiler only knows that the parameter is a `Shape' without knowing the exact kind of Shape it is. Furthermore suppose the `pick_and_drag(Shape*) function just mentioned was compiled on Tuesday, and on Wednesday you decide to add the Hexagon shape. Strange as it sounds, pick_and_drag() will still work with Hexagons, even though the Hexagon didn't even exist when pick_and_drag() was compiled!! (it's not really `amazing' once you understand how the C++ compiler does it -- but it's still very convenient!)

Q3: Who uses C++?

A: Lots and lots of companies and government sites. Lots.
Statistically, 20 to 30 people will consider themselves to be new C++
programmers before you finish reading the responses to these FAQs.

Q4: Are there any C++ standardization efforts underway?

A: Yes; ANSI (American) and ISO (International) groups are working closely with each other.

`X3J16' is the name of the ANSI-C++ committee.

`WG21' is the name of ISO's C++ standards group.

The committees are using the `ARM' as a base document:

> *Annotated C++ Reference Manual*, Ellis and Stroustrup, Addison/Wesley.
> ISBN 0-201-51459-1

The major players in the ANSI/ISO C++ standards process includes just about everyone:

AT&T, IBM, DEC, HP, Sun, MS, Borland, Zortech, Apple, OSF, <add your favorite here>, ... and a lot of users and smaller companies. About 70 people attend each ANSI C++ meeting. People come from USA, UK, Japan, Germany, Sweden, Denmark, France, ... (all have `local' committees sending official representatives and conducting `local' meetings).

Optimistically the standard might be finished by 1995-6 time frame (this is fast for a proper standards process).

Q5: Where can I ftp a copy of the latest ANSI-C++ draft standard?

A: You can't. ANSI standards and/or drafts are NOT available in machine readable form.

>>>>UPDATED 9/93
You can get a paper copy by sending a request to:

Standards Secretariat
CBEMA/X3
1250 I Street NW
Suite 200
Washington, DC 20005

Ask for the latest version of `Working Paper for Draft Proposed American National Standard for Information Systems -- Programming Language C++'. The last known phone number: 202-626-5738. The last known price is $25.

Q6: Is C++ backward compatible with ANSI-C?

A: Almost. C++ is as close as possible to compatible with ANSI-C but no closer. In practice, the major difference is that C++ requires prototypes, and that `f()' declares a function that takes no parameters, while ANSI-C rules state that `f()' declares a function that takes any number of parameters of any type. There are some very subtle differences as well, like the sizeof a char literal being equal to the sizeof a char (in ANSI-C, sizeof('x') is the sizeof an int). Structure `tags' are in the same namespace as other names in C++, but C++ has some warts to take care of backward compatibility here.

Q7: How long does it take to learn C++?

A: I and others teach standard industry `short courses' (for those not familiar with these, you pack a university semester course into one 40hr work-week), and have found them successful. However mastery takes experience, and there's no substitute for time. Laboratory time is essential for any OOP course, since it allows concepts to `gel'.

Generally people start out wondering why the company has devoted a full 5 days to something as trivial as another programming language. Then about half way through, they realize they're not being taught just a new syntax, but an entirely different way of thinking and programming and designing and . . . . Then they begin to feel dumb, since they can't quite grasp what is being said. Then they get mad and wonder why the course isn't taught in two or three weeks instead. Finally about Wednesday afternoon the lights go `clink', and their faces brighten, and they `get it'. By Friday, they've had numerous laboratory `experiments' and they've seen both sides of reusable components (both how to code *from* reuse, and how to code *for* reuse). It's different in every time I teach, but the `reuse' aspect is rewarding, since it has a large potential to improve software production's overall economics.

It takes 9 months to `master' C++/OOP. Less if there is already a body of experts and code that programmers have regular access to, more if there isn't a `good' general purpose C++ class library available.

Q8: What is a class?

A: A class defines a data type, much like a struct would be in C. In a CompSci sense, a type consists of two things: a set of values *and* a set of operations which operate on those values. Thus `int' all by itself isn't a true `type' until you add operations like `add two ints' or `int*int', etc. In exactly the same way, a `class' provides a set of (usually `public') operations, and a set of (usually non-public) data bits representing the abstract values that instances of the type can have. From a C language perspective, a `class' is a `struct' whose members default to `private'.

Q9: What is an object?

A: An object is a region of storage with associated semantics. After the declaration `int i;', we say that `i is an object of type int'. In C++/OOP, `object' is usually used to mean `an instance of a class'. Thus a class defines the behavior of possibly many objects (instances).

Q10: What is a reference?

A: A reference is an alias (an alternate name) for an object. It is frequently used for pass-by-reference; ex:

```
void swap(int& i, int& j)
{
  int tmp = i;
  i = j;
  j = tmp;
}

main()
{
  int x, y;
  //...
  swap(x,y);
}
```

Here `i' and `j' are aliases for main's `x' and `y' respectively. The effect is as if you used the C style pass-by-pointer, but the `&' is moved from the caller into the callee. Pascal enthusiasts will recognize this as a VAR param.

Q11: What happens if you assign to a reference?

A: Assigning to a reference changes the referred-to value, thus a ref is an `Lvalue' (something that can appear on the `L'eft-hand-side of an assignment statement) for the referred-to value. This insight can be pushed a bit farther by allowing references to be *returned*, thus allowing function calls on the left hand side of an assignment stmt.

Q12: How can you reseat a reference to make it refer to a different object?

A: Unlike a pointer, once a reference is bound to an object, it can NOT be `reseated' to another object. The reference itself isn't an object; you can't separate the reference from the referred-to-object. Ex: `&ref' is the address of the referred-to-object, not of the reference itself.

Q13: When should I use references, and when should I use pointers?

A: Old line C programmers sometimes don't like references since the reference semantics they provide isn't *explicit* in the caller's code. After a bit of C++ experience, however, one quickly realizes this `information hiding' is an asset rather than a liability. In particular, reuse-centered OOP tends to migrate the level of abstraction away from the language of the machine toward the language of the problem.

References are usually preferred over ptrs whenever you don't need `reseating' (see early question on `How can you reseat a reference'). This usually means that references are most useful in a class' public interface. References then 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. 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 ptr).

Q14: What are inline fns? What are their advantages? How are they declared?

A: An inline function is a function which gets textually inserted by the compiler, much like a macro. Like macros, performance is improved by avoiding the overhead of the call itself, and (especially!) by the compiler being able to optimize *through* the call (`procedural integration'). Unlike macros, arguments to inline fns are always evaluated exactly once, so the `call' is semantically like a regular function call only faster. Also unlike macros, argument types are checked and necessary conversions are performed correctly.

Beware that overuse of inline functions can cause code bloat, which can in turn have a negative performance impact in paging environments.

They are declared by using the `inline' keyword when the function is defined:

```
inline void f(int i, char c) { /*...*/ }   //an inline function
```

or by including the function definition itself within a class:

```
class X {
public:
  void f(int i, char c) { /*...*/ }   //inline function within a class
};
```

or by defining the member function as `inline' outside the class:

```
class X {
public:
  void f(int i, char c);
};
//...
inline void X::f(int i, char c) {/*...*/} //inline fn outside the class
```

Generally speaking, a function cannot be defined as `inline' after it has been called. Inline functions should be defined in a header file, with `outlined' functions appearing in a `.C' file (or .cpp, etc; see question on file naming conventions).

Q15: What is a constructor? Why would I ever use one?

A: Objects should establish and maintain their own internal coherence. The `maintaining' part is done by ensuring self-consistency is restored after any operation completes (ex: by incrementing the link count after adding a new link to a linked list).  The part about `establishing coherence' is the job of a constructor.

Constructors are like `init functions'; they build a valid object. The constructor turns a pile of incoherent arbitrary bits into a living object. Minimally it initializes any internally used fields that are needed, but it may also allocate resources (memory, files, semaphores, sockets, ...).

A constructor is like a `factory': it builds objects from dust.

`ctor' is a typical abbreviation for constructor.

Q16: How can I make a constructor call another constructor as a primitive?

A: You can't. Use an `init()' member function instead (often `private:').

Q17: What are destructors really for? Why would I ever use them?

A: Destructors are used to release any resources allocated by the object's constructor. Ex: a Lock class might lock a semaphore, and the destructor will release that semaphore. The usual `resource' being acquired in a constructor (and subsequently released in a destructor) is dynamically allocated memory.

`**dtor**' is a typical abbreviation for destructor

Q18: What is operator overloading?

A: Operator overloading allows the basic C/C++ operators to have user-defined meanings on user-defined types (classes). They are syntactic sugar for equivalent function calls; ex:

```
class X {
  //...
public:
  //...
};

X add(X, X);      //a top-level function that adds two X's
X mul(X, X);      //a top-level function that multiplies two X's

X f(X a, X b, X c)
{
  return add(add(mul(a,b), mul(b,c)), mul(c,a));
}
```

Now merely replace `add' with `operator+' and `mul' with `operator*':

```
X operator+(X, X);      //a top-level function that adds two X's
X operator*(X, X);      //a top-level function that multiplies two X's

X f(X a, X b, X c)
{
  return a*b + b*c + c*a;
}
```

Q19: 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 Vec {
  int data[100];
public:
  int& elem(unsigned i) { if (i>99) error(); return data[i]; }
};

main()
{
  Vec v;
  v.elem(10) = 42;
  v.elem(12) += v.elem(13);
}
```

Now simply replace `elem' with `operator[]':

```
class Vec {
  int data[100];
public:
  int& operator[](unsigned i) { if (i>99) error(); return data[i]; }
};   //^^^^^^^^^^--formerly `elem'

main()
{
  Vec v;
  v[10] = 42;
  v[12] += v[13];
}
```

Q20: Can I create a `**' operator for `to-the-power-of' operations?

A: No.

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 doubt the wisdom of this approach, consider the following code:

```
x = y ** z;
```

Looks like your power operator? Nope. z may be a ptr, so this is actually:

```
x = y * (*z);
```

Lexical analysis groups characters into tokens at the lowest level of the compiler's operations, so adding new operators would present an implementation nightmare (not to mention the increased maintenance cost to read the code!).

Besides, operator overloading is just syntactic sugar for function calls. It does not add fundamental power to the language (although this particular syntactic sugar can be very sweet, it is not fundamentally necessary). I suggest you overload `pow(base,exponent)', for which a double precision version is provided by the ANSI-C <math.h> library.

By the way: operator^ looks like a good candidate for to-the-power-of, but it has neither the proper precedence nor associativity.

Q21: What is a `friend'?

A: Friends can be either functions or other classes. The class grants friends access privileges. Normally a developer has political and technical control over both the class, its members, and its friends (that way you avoid political problems when you want to update a portion, since you don't have to get permission from the present owner of the other piece(s)).

Q22: Do `friends' violate encapsulation?

A: Friends can be looked at three ways: (1) they are not class members and they therefore violate encapsulation of the class members by their mere existence, (2) a class' friends are absorbed into that class' encapsulation barrier, and (3) any time anyone wants to do anything tricky they textedit the header file and add a new friend so they can get right in there and fiddle 'dem bits.

No one argues that (3) is a Good Thing, and for good reasons. The arguments for (1) always boil down to the rather arbitrary and somewhat naive view that a class' member functions `should' be the *only* functions inside a class' encapsulation barrier. I have not seen this view bear fruit by enhancing software quality. On the other hand, I have seen (2) bear fruit by lowering the *overall* coupling in a software system. Reason: friends can be used as `liaisons' to provide safe, screened access for the whole world, perhaps in a way that the class syntactically or semantically isn't able to do for itself.

Conclusion: friend functions are merely a syntactic variant of a class' public access functions. When used in this manner, they don't violate encapsulation any more than a member function violates encapsulation. Thus a class' friends and members *are* the encapsulation barrier, as defined by the class itself.

I've actually seen the `friends always violate encapsulation' view *destroy* encapsulation: programmers who have been taught that friends are inherently evil want to avoid them, but they have another class or fn that needs access to some internal detail in the class, so they provide a member fn which exposes the class' internal details to the PUBLIC! Private decisions should stay private, and only those inside your encapsulation barrier (your members, friends, and [for `protected' things] your subclasses) should have access.

Q23: What are some advantages/disadvantages of using friends?

A: The advantage of using friends is generally syntactic. Ie: both a member fn and a friend are equally privileged (100% vested), but a friend function can be called like f(obj), where a member is called like obj.f(). When it's not for syntactic reasons (which is not a `bad' reason -- making an abstraction's syntax more readable lowers maintenance costs!), friends are used when two or more classes are designed to be more tightly coupled than you want for `joe public' (ex: you want to allow class `ListIter' to have more privilege with class `List' than you want to give to `main()').

Friends have three disadvantages. The first disadvantage is that they add to the global namespace. In contrast, the namespace of member functions is buried within the class, reducing the chance for namespace collisions for functions.

The second disadvantage is that they aren't inherited. That is, the `friendship privilege' isn't inherited. This is actually an advantage when it comes to encapsulation. Ex: I may declare you as my friend, but that doesn't mean I trust your kids.

The third disadvantage is that they don't bind dynamically. Ie: they don't respond to polymorphism. There are no virtual friends; if you need one, have a friend call a hidden (usually `protected:') virtual member fn. Friends that take a ptr/ref to a class can also take a ptr/ref to a publically derived class object, so they act as if they are inherited, but the friendship *rights* are not inherited (the friend of a base has no special access to a class derived from that base).

Q24: What does it mean that `friendship is neither inherited nor transitive'?

A: This is speaking of the access privileges granted when a class declares a friend.

The access privilege of friendship is not inherited:
 * I may trust you, but I don't necessarily trust your kids.
 * My friends aren't necessarily friends of my kids.
 * Class `Base' declares f() to be a friend, but f() has no special access rights with class `Derived'.

The access privilege of friendship is not transitive:
 * I may trust you, and you may trust Sam, but that doesn't necessarily mean that I trust Sam.
 * A friend of a friend is not necessarily a friend.

Q25: When would I use a member function as opposed to a friend function?

A: Use a member when you can, and a friend when you have to.

Like in real life, my family members have certain privileges that my friends do not have (ex: my family members inherit from me, but my friends do not, etc). To grant privileged access to a function, you need either a friend or a member; there is no additional loss of encapsulation one way or the other. Sometimes friends are syntactically better (ex: in class `X', friend fns allow the `X' param to be second, while members require it to be first). Another good use of friend functions are the binary infix arithmetic operators. Ex: `aComplex + aComplex' probably should be defined as a friend rather than a member, since you want to allow `aFloat + aComplex' as well (members don't allow promotion of the left hand arg, since that would change the class of the object that is the recipient of the member function invocation).

Q26: How can I provide printing for a `class X'?

A: Provide a friend operator<<:

```
class X {
public:
  friend ostream& operator<< (ostream& o, const X& x)
    { return o << x.i; }
  //...
private:
  int i;    //just for illustration
};
```

We use a friend rather than a member since the `X' parameter is 2nd, not 1st. Input is similar, but the signature is:

```
istream& operator>> (istream& i, X& x);  //not `const X& x' !!
```

Q27: Why should I use <iostream.h> instead of the traditional <stdio.h>?

A: See next question.

Q28: Printf/scanf weren't broken; why `fix' them with ugly shift operators?

A: The overloaded shift operator syntax is strange at first sight, but it quickly grows on you. However syntax is just syntax; the real issues are deeper. Printf is arguably not broken, and scanf is perhaps livable despite being error prone, however both are limited with respect to what C++ I/O can do. C++ I/O (left/right shift) is, relative to C (printf/scanf):

* type safe -- type of object being I/O'd is known statically by the compiler rather than via dynamically tested '%' fields
* less error prone -- redundant info has greater chance to get things wrong C++ I/O has no redundant '%' tokens to get right
* faster -- printf is basically an `interpreter' of a tiny language whose constructs mainly include '%' fields. the proper low-level routine is chosen at runtime based on these fields. C++ I/O picks these routines statically based on actual types of the args
* extensible -- perhaps most important of all, the C++ I/O mechanism is extensible to new user-defined data types (imagine the chaos if everyone was simultaneously adding new incompatible '%' fields to printf and scanf?!). Remember: we want to make user-defined types (classes) look and act like `built-in' types.
* subclassable -- ostream and istream (the C++ replacements for FILE*) are real classes, and hence subclassable. This means you can have other user defined things that look and act like streams, yet that do whatever strange and wonderful things you want. You automatically get to use the zillions of lines of I/O code written by users you don't even know, and they don't need to know about your `extended stream' class. Ex: you can have a `stream' that writes to a memory area (incore formatting provided by the standard class `strstream'), or you could have it use the stdio buffers, or [you name it...].

Q29: Does `delete ptr' delete the ptr or the pointed-to-data?

A: The pointed-to-data.

When you read `delete p', say to yourself `delete the thing pointed to by p'. One could argue that the keyword is misleading, but the same abuse of English occurs when `free'ing the memory pointed to by a ptr in C:

```
free(ptr);  /* why not `free_the_stuff_pointed_to_by(p)' ?? */
```

Q30: Can I free() ptrs alloc'd with `new' or `delete' ptrs alloc'd w/ malloc()?

A: No. You should not mix C and C++ heap management.

Q31: Why should I use `new' instead of trustworthy old malloc()?

A: malloc() doesn't call constructors, and free() doesn't call destructors. Besides, malloc() isn't type safe, since it returns a `void*' rather than a ptr of the right type (ANSI-C punches a hole in its typing system to make it possible to use malloc() without pointer casting the return value, but C++ closes that hole). Besides, `new' is an operator that can be overridden by a class, while `malloc' is not overridable on a per-class basis (ie: even if the class doesn't have a constructor, allocating via malloc might do inappropriate things if the freestore operations have been overridden).

Q32: Why doesn't C++ have a `realloc()' along with `new' and `delete'?

Q32: Why doesn't C++ have a `realloc()' along with `new' and `delete'?

A: Because realloc() does *bitwise* copies (when it has to copy), which will tear most C++ objects to shreds. C++ objects know how to copy themselves. They use their own copy constructor or assignment operator (depending on whether we're copying into a previously unused space [copy-ctor] or a previous object [assignment op]).

Moral: never use realloc() on objects of a class. Let the class copy its own objects.

Q33: How do I allocate / unallocate an array of things?

A: Use new[] and delete[]:

```
Thing* p = new Thing[100];
//...
delete [] p;          //older compilers require you to use `delete [100] p'
```

Any time you allocate an array of things (ie: any time you use the `[...]' in the `new' expression) you *!*MUST*!* use the `[]' in the `delete' statement.

The fact that there is no syntactic difference between a ptr to a thing and a ptr to an array of things is an artifact we inherited from C.

Q34: What if I forget the `[]' when `delete'ing array allocated via `new X[n]'?

A: Life as we know it suddenly comes to a catastrophic end.

It is the programmer's --not the compiler's-- responsibility to get the connection between new[] and delete[] 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.

Q35: How can I handle a constructor that fails?

A: Constructors (ctors) do not return any values, so no returned error code is possible. The best way to handle failure is therefore to `throw' an exception.

If your compiler doesn't yet support exceptions, several possibilities remain. The simplest is to put the object itself into a `half baked' state by setting an internal status bit. Naturally there should be a query (`inspector') method to check this bit, allowing clients to discover whether they have a live object. Other member functions should check this bit, and either do a no-op (or perhaps something more obnoxious such as `abort()') if the object isn't really alive. Check out how the iostreams package handles attempts to open nonexistent/illegal files for an example of prior art.

Q36: How can I compile-out my debugging print statements?

A: This will NOT work, since comments are parsed before the macro is expanded:

```
#ifdef DEBUG_ON
  #define  DBG
#else
  #define  DBG  //
#endif
DBG cout << foo;
```

This is the simplest technique:

```
#ifdef DEBUG_ON
  #define  DBG(anything)  anything
#else
  #define  DBG(anything)  /*nothing*/
#endif
```

Then you can say:

```
//...
DBG(cout << "the value of foo is " << foo << '\n');
//                                                ^-- `;' outside ()
```

Any commas in your `DBG()' statement must be enclosed in a `()':

```
DBG(i=3, j=4);    //<---- C-preprocessor will generate error message
DBG(i=3; j=4);    //<---- ok
```

There are also more complicated techniques that use variable argument lists, but these are primarily useful for `printf()' style (see question on the pros and cons of <iostream.h> as opposed to <stdio.h> for more).

Q37: What is `const correctness'?

A: A program is `const correct' if it never mutates a constant object. This is achieved by using the keyword `const'. Ex: if you pass a String to a function `f()', and you wish to prohibit `f()' from modifying the original String, you can either pass by value:

```
     void  f(      String  s)  { /*...*/ }
```

or by constant reference:

```
     void  f(const String& s   )  { /*...*/ }
```

or by constant pointer:

```
     void  f(const String* sptr)  { /*...*/ }
```

but *not* by non-const ref:

```
     void  f(      String& s   )  { /*...*/ }
```

nor by non-const pointer:

```
     void  f(      String* sptr)  { /*...*/ }
```

Attempted changes to `s' within a fn that takes a `const String&' are flagged as compile-time errors; neither run-time space nor speed is degraded.

Q38: Is `const correctness' a good goal?

A: Declaring the `constness' of a parameter is just another form of type safety. It is almost as if a constant String, for example, `lost' its various mutative operations. If you find type safety helps you get systems correct (especially large systems), you'll find const correctness helps also.

Short answer: yes, const correctness is a good goal.

Q39: Is `const correctness' tedious?

A: Type safety requires you to annotate your code with type information. In theory, expressing this type information isn't necessary -- witness untyped languages as an example of this. However in practice, programmers often know in their heads a lot of interesting information about their code, so type safety (and, by extension, const correctness) merely provide structured ways to get this information into their keyboards.

Short answer: yes, const correctness is tedious.

Q40: Should I try to get things const correct `sooner' or `later'?

A: Back-patching const correctness is *very* expensive. Every `const' you add `over here' requires you to add four more `over there'. The snowball effect is magnificent -- unless you have to pay for it. Long about the middle of the process, someone stumbles on a function that needs to be const but can't be const, and then they know why their system wasn't functioning correctly all along. This is the benefit of const correctness, but it should be installed from the beginning.

Short answer: CONST CORRECTNESS SHOULD NOT BE DONE RETROACTIVELY!!

Q41: What is a `const member function'?

A: A const member function is a promise to the caller not to change the object. Put the word `const' after the member function's signature; ex:

```
class X {
  //...
  void f() const;
};
```

Some programmers feel this should be a signal to the compiler that the raw bits of the object's `struct' aren't going to change, others feel it means the *abstract* (client-visible) state of the object isn't going to change. C++ compilers aren't allowed to assume the bitwise const, since a non-const alias could exist which could modify the state of the object (gluing a `const' ptr to an object doesn't promise the object won't change; it only promises that the object won't change **via that pointer**).

I talked to Jonathan Shopiro at the C++AtWork conference, and he confirmed that the above view has been ratified by the ANSI-C++ standards board. This doesn't make it a `perfect' view, but it will make it `the standard' view.

See the next few questions for more.

Q42: What is an `inspector'? What is a `mutator'?

A: An inspector inspects and a mutator mutates. These different categories of member fns are distinguished by whether the member fn is `const' or not.

Q43: What is `casting away const in an inspector' and why is it legal?

A: In current C++, const member fns are allowed to `cast away the const-ness of the "this" ptr'. Programmers use (some say `misuse') this to tickle internally used counters, cache values, or some other non-client-visible change. Since C++ allows you to use const member fns to indicate the abstract/meaning-wise state of the object doesn't change (as opposed to the concrete/bit-wise state), the `meaning' of the object shouldn't change during a const member fn.

Those who believe `const' member fns shouldn't be allowed to change the bits of the struct itself call the `abstract const' view `Humpty Dumpty const' (Humpty Dumpty said that words mean what he wants them to mean). The response is that a class' public interface *should* mean exactly what the class designer wants it to mean, in Humpty Dumpty's words, `nothing more and nothing less'. If the class designer says that accessing the length of a List doesn't change the List, then one can access the length of a `const' List (even though the `len()' member fn may internally cache the length for future accesses).

Some proposals are before the ANSI/ISO C++ standards bodies to provide syntax that allows individual data members to be designated as `can be modified in a const member fn' using a prefix such as `~const'. This would blend the best of the `give the compiler a chance to cache data across a const member fn', but only if aliasing can be solved (see next question).

Q44: But doesn't `cast away const' mean lost optimization opportunities?

A: If the object is constructed in the scope of the const member fn invocation, and if all the non-const member function invocations between the object's construction and the const member fn invocation are statically bound, and if every one of these invocations is also `inline'd, and if the ctor itself is `inline', and if any member fns the ctor calls are inline, then the answer is `Yes, the soon-to-be-standard interpretation of the language would prohibit a very smart compiler from detecting the above scenario, and the register cache would be unnecessarily flushed'. The reader should judge whether the above scenario is common enough to warrant a language change which would break existing code.

Q45: What is inheritance?

A: Inheritance is what separates abstract data type (ADT) programming from OOP. It is not a `dark corner' of C++ by any means. In fact, everything discussed so far could be simulated in your garden variety ADT programming language (ex: Ada, Modula-2, C [with a little work], etc). Inheritance and the consequent (subclass) polymorphism are the two big additions which separate a language like Ada from an object-oriented programming language.

Q46: Ok, ok, but what is inheritance?

A: Human beings abstract things on two dimensions: part-of and kind-of. We say that a Ford Taurus is-a-kind-of-a Car, and that a Ford Taurus has parts such as Engine, Tire, etc. The part-of hierarchy has been a first class part of software since the ADT style became relevant, but programmers have had to whip up their own customized techniques for simulating kind-of (usually in an ad hoc manner). Inheritance changes that; it adds `the other' major dimension of decomposition.

An example of `kind-of decomposition', consider the genus/species biology charts. Knowing the internal parts of various fauna and flora is important for certain applications, but knowing the groupings (kinds, categories) is equally important.

Q47: How do you express inheritance in C++?

A: By the `: public' syntax:

```
class Car : public Vehicle {
        //^^^^^^^^^---- `: public' is pronounced `is-a-kind-of-a'
  //...
};
```

We state the above relationship in several ways:
* Car is `a kind of a' Vehicle
* Car is `derived from' Vehicle
* Car is `a specialized' Vehicle
* Car is the `subclass' of Vehicle
* Vehicle is the `base class' of Car
* Vehicle is the `superclass' of Car (this not as common in the C++ community)

Q48: What is `incremental programming'?

A: In addition to being an abstraction mechanism that makes is-a-kind-of relationships explicit, inheritance can also be used as a means of `incremental programming'. A derived class inherits all the representation (bits) of its base class, plus all the base class' mechanism (code). Another device (virtual functions, described below) allows derived classes to selectively override some or all of the base class' mechanism (replace and/or enhance the various algorithms).

This simple ability is surprisingly powerful: it effectively adds a `third dimension' to programming. After becoming fluent in C++, most programmers find languages like C and Ada to be `flat' (a cute little book, `Flatland', aptly describes those living in a two dimensional plane, and their disbelief about a strange third dimension that is somehow neither North, South, East nor West, but is `Up').

As a trivial example, suppose you have a Linked List that is too slow, and you wish to cache its length. You could `open up' the List `class' (or `module'), and modify it directly (which would certainly be appropriate for such a simple situation), but suppose the List's physical size is critical, and some important client cannot afford to add the extra machine word to every List. Another option would be to textually copy the List module and modify the copy, but this increases the amount of code that must be maintained, and also presumes you have access to the internal source code of the List module. The OO solution is to realize that a List that caches its length is-a-kind-of-a List, so we inherit:

```
class FastList : public List {
public:
  //override operations so the cache stays `hot'
protected:
  int length;     //cache the length here
};
```

Q49: Should I pointer-cast from a derived class to its base class?

A: The short answer: yes -- you don't even need the `cast'.

Long answer: a derived class is a specialized version of the base class (`Derived is-a-kind-of-a Base'). The upward conversion is perfectly safe, and happens all the time (a ptr to a Derived is in fact pointing to a [specialized version of a] Base):

```
void f(Base* base_ptr);
void g(Derived* derived_ptr) { f(derived_ptr); }    //perfectly safe; no cast
```

(note that the answer to this question assumes we're talking about `public' derivation; see below on `private/protected' inheritance for `the other kind').

Q50: Derived* --> Base* works ok; why doesn't Derived** --> Base** work?

A: A C++ compiler will allow a Derived* to masquerade as a Base*, since a Derived object is a kind of a Base object. However passing a Derived** as a Base** (or otherwise trying to convert a Derived** to a Base**) is (correctly) flagged as an error.

An array of Deriveds is-NOT-a-kind-of-an array of Bases. I like to use the following example in my C++ training sessions:

`A Bag of Apples is *NOT* a Bag of Fruit'

Suppose a `Bag<Apple>' could be passed to a function taking a Bag<Fruit> such as `f(Bag<Fruit>& b)'. But `f()' can insert *any* kind of Fruit into the Bag. Imagine the surprise on the caller's face when he gets the Bag back only to find it has a Banana in it!

Here's another example I use:

A ParkingLot of Car is-NOT-a-kind-of-a ParkingLot of Vehicle (otherwise you could pass a ParkingLot<Car>* as a ParkingLot<Vehicle>*, and the called fn could park an EighteenWheeler in a ParkingLot designed for Cars!)

These improper things are violations of `contravariance' (that's the scientific glue that holds OOP together). C++ enforces contravariance, so you should trust your compiler at moments like these. Contravariance is more solid than our fickle intuition.

Q51: Does array-of-Derived is-NOT-a-kind-of array-of-Base mean arrays are bad?

A: Yes, `arrays are evil' (jest kidd'n :-).

There's a very subtle problem with using raw built-in arrays. Consider this:

```
void f(Base* array_of_Base)
{
  array_of_Base[3].memberfn();
}

main()
{
  Derived array_of_Derived[10];
  f(array_of_Derived);
}
```

This is perfectly type-safe, since a D* is-a B*, but it is horrendously evil, since Derived might be larger than Base, so the array index in f() not only isn't type safe, it's not even going to be pointing at a real object! In general it'll be pointing somewhere into the innards of some poor D. The fundamental problem here is that C++ cannot distinguish a ptr-to-a-thing from a ptr-to-an-array-of-things (witness the required `[]' in `delete[]' when deleting an array as another example of how these different kinds of ptrs are actually different). Naturally C++ `inherited' this feature from C.

This underscores the advantage of using an array-like *class* instead of using a raw array (the above problem would have been properly trapped as an error if we had used a `Vec<T>' rather than a `T[]'; ex: you cannot pass a Vec<Derived> to `f(Vec<Base>& v)').

Q52: What is a `virtual member function'?

A: A virtual member function is a member fn preceded by the keyword `virtual'. It has the effect of allowing derived classes to replace the implementation of the fn. Furthermore the replacement is always called whenever the object in question is actually of the derived class. The impact is that algorithms in the base class can be replaced in the derived class without affecting the operation of the base class. The replacement can be either full or partial, since the derived class operation can invoke the base class version if desired.

This is discussed further in the next topic Q53..

Q53: What is dynamic dispatch? Static dispatch?

A: In the following discussion, `ptr' means either a pointer or a reference.

When you have a ptr to an object, there are two distinct types in question: the static type of the ptr, and the dynamic type of the pointed-to object (the object may actually be of a class that is derived from the class of the ptr).

The `legality' of the call is checked based on the static type of the ptr,

which gives us static type safety (if the type of the ptr can handle the member fn, certainly the pointed-to object can handle it as well, since the pointed-to object is of a class that is derived from the ptr's class).

Suppose ptr's type is `List' and the pointed-to object's type is `FastList'. Suppose the fn `len()' is provided in `List' and overridden in `FastList'.  The question is: which function should actually be invoked: the function attached to the pointer's type (`List::len()') or the function attached to the object itself (`FastList::len()')?

If `len()' is a virtual function, as it would be in the above case, the fn attached to the object is invoked.  This is called `dynamic binding', since the actual code being called is determined dynamically (at run time).

On the other hand, if `len()' were non-virtual, the dispatch would be resolved statically to the fn attached to the ptr's class.

Q54: Can I override a non-virtual fn?

A: Yes but you shouldn't. The only time you should do this is to get around the `hiding rule' (see below, and ARM sect.13.1), and the overridden definition should be textually identical to the base class' version.

The above advice will keep you out of trouble, but it is a bit too strong. Experienced C++ programmers will sometimes override a non-virtual fn for efficiency, and will provide an alternate implementation which makes better use of the derived class' resources. However the client-visible effects must be *identical*, since non-virtual fns are dispatched based on the static type of the ptr/ref rather than the dynamic type of the pointed-to/referenced object.

Q55: Why do I get the warning "Derived::foo(int) hides Base::foo(double)"?

A: A member function in a derived class will *hide* all member functions of the same name in the base class, *not* overload them, even if Base::foo(double) is virtual (see ARM 13.1). This is done because it was felt that programmers would, for example, call a_derived.foo(1) and expect Derived::foo(double) to be called. If you define any member function with the name `foo' in a derived class, you must redefine in class Derived all other Base::foo()'s that you wish to allow access from a Derived object (which generally means all of them; you should [generally] *not* try to hide inherited public member functions since it breaks the `conformance' of the derived class with respect to the base class).

```
class Base {
public:
  void foo(int);
};

class Derived : public Base {
public:
  void foo(double);
  void foo(int i) { Base::foo(i); }   // <-- override it with itself
};
```

Q56: Can I `revoke' or `hide' public member fns inherited from my base class?

A: Never never never do this. Never. NEVER!

This is an all-too-common design error. It usually stems from muddy thinking (but sometimes it stems from a very difficult design that doesn't seem to yield anything elegant).

Q57: Is a `Circle' a kind-of an `Ellipse'?

A: Depends on what you claim an Ellipse can do. Ex: suppose Ellipse has a `scale(x,y)' method, which is meaningless for Circle.

There are no easy options at this point, but the worst of all possible worlds is to keep muddling along and hope that no one stubs their toes over the bad design (if we're serious about reuse, we should fix our mistakes rather than leave them to a future generation). If an Ellipse can do something a Circle can't, a Circle can't be a kind of Ellipse. Should there be any other relationship between Circle and Ellipse? Here are two reasonable options: * make Circle and Ellipse completely unrelated classes. * derive Circle and Ellipse from a base class representing `Ellipses that can't *necessarily* perform an unequal-scale operation'.

In the first case, Ellipse could be derived from class `AsymmetricShape' (with scale(x,y) being introduced in AsymmetricShape), and Circle should be derived from `SymmetricShape', which has a scale(factor) member fn.

In the second case, we could create class `Oval' that has only an equal scale operation, then derive both `Ellipse' and `Circle' from Oval, where Ellipse --but not Circle-- adds the unequal scale operation (see the `hiding rule' for a caveat if the same method name `scale' is used for both unequal and equal scale operations).

In any event, we could create an operation to create an Ellipse whose size et al are the same as a given Circle, but this would be a constructive operation (ie: it would create a brand new object, like converting an int to a float, but unlike passing a reference to a Circle as if it were a ref to an Ellipse).

```
Example:   class Ellipse : public Oval {
           public:       ^^^^^^^^^^^^^---- or whatever
             Ellipse(const Circle& circle);
             //...
           };
Or:        class Circle /*...*/ {
           public:
             operator Ellipse() const;
             //...
           };
```

Q58: Are there other options to the `Circle is/isnot kind-of Ellipse' dilemma?

A: There appear to be 2 other options (but read below for why these are poor): * redefine Circle::scale(x,y) to throw an exception or call `abort()'. * redefine Circle::scale(x,y) to be a no-op, or to scale both dimensions by the average of the parameters (or some other arbitrary value).

Throwing an exception will `surprise' clients. You claimed that a Circle was actually a kind of an Ellipse, so they pass a Circle off to `f(Ellipse& e)'. The author of this function read the contract for Ellipse very carefully, and scale(x,y) is definitely allowed. Yet when f() innocently calls e.scale(5,3), it kills him! Conclusion: you lied; what you gave them was distinctly *not* an Ellipse.

In the second case, you'll find it to be very difficult to write a meaningful semantic specification (a `contract') for Ellipse::scale(x,y). You'd like to be able to say it scales the x-axis by `x' and the y-axis by `y', but the best you can say is `it may do what you expect, or it may do nothing, or it may scale both x and y even if you asked it to only scale the x (ex: `scale(2,1)'). Since you've diluted the contract into dribble, the client can't rely on any meaningful behavior, so the whole hierarchy begins to be worthless (it's hard to convince someone to use an object if you have to shrug your shoulders when asked what the object does for them).

Q59: Why can't I access `private' things in a base class from a derived class?

A: Derived classes do not get access to private members of a base class. This effectively `seals off' the derived class from any changes made to the private members of the base class.

Q60: What's the difference between `public:', `private:', and `protected:'?

A: `Private' is discussed in the previous section, and `public' means `anyone can access it'. The third option, `protected', makes a member (either data member or member fn) accessible to subclasses.

Thus members defined in the `private:' section of class X are accessible only to the member functions and friends of class X; members defined in the `public:' section are accessible by everyone; `protected:' members are accessible by members fns and friends of class X, as well as member fns of subclasses of X.

Q61: How can I protect subclasses from breaking when I change internal parts?

A: You can make your software more resilient to internal changes by realizing a class has two distinct interfaces for two distinct sets of clients: * its `public:' interface serves unrelated classes  * its `protected:' interface serves derived classes

A class that is intended to have a long and happy life can hide its physical bits in its `private:' part, then put `protected:' inline access functions to these data. The private bits can change, but if the protected access fns are stable, subclasses (ie: derived classes) won't break (though they'll need to be recompiled after a change to the base class).

Q62: Why does base ctor get *base*'s virtual fn instead of the derived version?

Ie: when constructing an obj of class `Derived', Base::Base() invokes `virt()'. `Derived::virt()' exists (an override of `Base::virt()'), yet `Base::virt()' gets control rather than the `Derived' version; why?

A: A constructor turns raw bits into a living object. Until the ctor has finished, you don't have a complete `object'. In particular, while the base class' ctor is working, the object isn't yet a Derived class object, so the call of the base class' virtual fn defn is correct.

Similarly dtors turn a living object into raw bits (they `blow it to bits'), so the object is no longer a Derived during Base's dtor. Therefore the same thing happens: when Base::~Base() calls `virt()', Base::virt() gets control, not the Derived::virt() override. (Think of what would happen if the Derived fn touched a subobject from the Derived class, and you'll quickly see the wisdom of the approach).

Q63: Does a derived class dtor need to explicitly call the base destructor?

A: No, you never need to explicitly call a dtor (where `never' means `rarely'). ie: you only have to have an explicit dtor call in rather esoteric situations such as destroying an object created by the `placement new operator'. In the usual case, a derived class' dtor (whether you explicitly define one or not) automatically invokes the dtors for subobjects and base class(es). Subobjects are destroyed immediately after the derived class' destructor body (`{...}'), and base classes are destroyed immediately after subobjects. Subobjects are destroyed from bottom to top in the lexical order they appear within a class, and base classes from right to left in the order of the base-class-list.

Q64: How do you express `private inheritance'?

A: When you use `: private' instead of `: public'. Ex:

```
class Foo : private Bar {
  //...
};
```

Q65: How are `private derivation' and `containment' similar? dissimilar?

A: Private derivation can be thought of as a syntactic variant of containment (has-a). Ex: it is NOT true that a privately derived is-a-kind-of-a Base:

With private derivation:

```
     class Car : private Engine {/*...*/};     //a Car is NOT a-kind-of
Engine
```

Similarly:

```
     class Car { Engine e; /*...*/ };    //normal containment
```

There are several similarities between these two forms of containment:
* in both cases there is exactly one Engine subobject contained in a Car
* in neither case can clients (outsiders) convert a Car* to an Engine*

There are also several distinctions:
* the second form is needed if you want to contain several subobjects
* the first form can introduce unnecessary multiple inheritance
* the first form allows members of Car to convert a Car* to an Engine*
* the first form allows access to the `protected' members of the base class
* the first form allows Car to override Engine's virtual functions.

Private inheritance is almost always used for the last item: to gain access into the `protected:' members of the base class.

Q66: Should I pointer-cast from a `privately' derived class to its base class?

A: The short answer: no, but yes too (better read the long answer!)

>From `inside' the privately derived class (ie: in the body of members or friends of the privately derived class), the relationship to the base class is known, and the upward conversion from PrivatelyDer* to Base* (or PrivatelyDer& to Base&) is safe and doesn't need a cast.

>From `outside' the privately derived class, the relationship to `Base' is a `private' decision of `PrivatelyDer', so the conversion requires a cast. Clients should not exercise this cast, since private derivation is a private implementation decision of the privately derived class, and the coercion will fail after the privately derived class privately chooses to change this private implementation decision.

Bottom line: only a class and its friends have the right to convert a ptr to a derived class into a ptr to its private base class. They don't need a cast, since the relationship with the base class is accessible to them. No one else can convert such ptrs without pointer-casts, so no one else should.

Q67: Should I pointer-cast from a `protected' derived class to its base class?

A: Protected inheritance is similar to private inheritance; the answer is `no'.

```
class Car : protected Engine {/*...*/};   //protected inheritance
```

In `private' inheritance, only the class itself (and its friends) can know about the relation to the base class (the relationship to the base class is a `private' decision). In protected inheritance, the relationship with the base class is a `protected' decision, so *subclasses* of the `protectedly' derived class can also know about and exploit this relationship.

This is a `for better *and* for worse' situation: future changes to `protected' decisions have further consequences than changing a private decision (in this case, the class, its friends, *and* subclasses, sub- sub- classes, etc, all need to be examined for dependencies upon the relationship to the base class). However it is also for better, in that subclasses have the ability to exploit the relationship.

The existence of protected inheritance in C++ is debated in some circles.

Q68: What are the access rules with `private' and `protected' inheritance?

A: Take these classes as examples:

```
class B { /*...*/ };
class D_priv : private   B { /*...*/ };
class D_prot : protected B { /*...*/ };
class D_publ : public    B { /*...*/ };
class Client { B b; /*...*/ };
```

Public and protected parts of B are `private' in D_priv, and are `protected' in D_prot. In D_publ, public parts of B are public (D_prot is-a-kind-of-a B), and protected parts of B remain protected in D_publ. Naturally *none* of the subclasses can access anything that is private in B. Class `Client' can't even access the protected parts of B (ie: it's `sealed off').

It is often the case that you want to make some but not all inherited member functions public in privately/protectedly derived classes. Ex: to make member fn B::f(int,char,float) public in D_prot, you would say:

```
class D_prot : protected B {
  //...
public:
  B::f;    //note: not  B::f(int,char,float)
};
```

There are limitations to this technique (can't distinguish overloaded names, and you can't make a feature that was `protected' in the base `public' in the derived). Where necessary, you can get around these by a call-through fn:

```
class D_prot : protected B {
public:
  short f(int i, char c, float f) { return B::f(i,c,f); }
};
```

Q69: Do most C++ programmers use containment or private inheritance?

A: Short answer: generalizations are always wrong (that's a generalization :-).

The long answer is another generalization: most C++ programmers use regular containment (also called `composition' or `aggregation') more often than private inheritance. The usual reason is that they don't *want* to have access to the internals of too many other classes.

Private inheritance is not evil; it's just more expensive to maintain, since it increases the number of classes that have access to `internal' parts of other classes (coupling). The `protected' parts of a class are more likely to change than the `public' parts.

Q70: What's the big deal of separating interface from implementation?

A: Separating interface from implementation is a key to reusable software. Interfaces are a company's most valuable resources. Designing an interface takes longer than whipping together a concrete class which fulfills that interface. Furthermore interfaces require the resources of more expensive people (for better and worse, most companies separate `designers' from `coders'). Since they're so valuable, they should be protected from being tarnished by data structures and other artifacts of the implementation (any data structures you put in a class can never be `revoked' by a derived class, which is why you want to `separate' the interface from the implementation).

Q71: How do I separate interface from implementation in C++ (like Modula-2)?

A: Short answer: use an ABC (see next question for what an ABC is).

Q72: What is an ABC (`abstract base class')?

A: An ABC corresponds to an abstract concept. If you asked a Mechanic if he repaired Vehicles, he'd probably wonder what *kind* of Vehicle you had in mind. Chances are he doesn't repair space shuttles, ocean liners, bicycles, and volkswaggon beetles too. The problem is that the term `Vehicle' is an abstract concept; you can't build one until you know what kind of vehicle to build. In C++, you'd make Vehicle be an ABC, with Bicycle, SpaceShuttle, etc, being subclasses (an OceanLiner is-a-kind-of-a Vehicle).

In real-world OOP, ABCs show up all over the place. Technically, an ABC is a class that has one or more pure virtual member functions (see next question). You cannot make an object (instance) of an ABC.

Q73: What is a `pure virtual' member function?

A: Some member functions exist in concept, but can't have any actual defn. Ex: Suppose I asked you to draw a Shape at location (x,y) that has size 7.2. You'd ask me `what kind of shape should I draw', since circles, squares, hexagons, etc, are drawn differently. In C++, we indicate the existence of the `draw()' method, but we recognize it can only be defined in subclasses:

```
class Shape {
public:
  virtual void draw() const = 0;
  //...                      ^^^--- `=0' means it is `pure virtual'
};
```

This pure virtual makes `Shape' an ABC. The `const' says that invoking the `draw()' method won't change the Shape object (ie: it won't move around on the screen, change sizes, etc). If you want, you can think of it as if the code were at the NULL pointer.

Pure virtuals allow you to express the idea that any actual object created from a [concrete] class derived from the ABC *will* have the indicated member fn, but we simply don't have enough information to actually *define* it yet. They allow separation of interface from implementation, which ultimately allows functionally equivalent subclasses to be produced that can `compete' in a free market sense (a technical version of `market driven economics').

Q74: How can I provide printing for an entire hierarchy rooted at `class X'?

A: Provide a friend operator<< that calls a protected virtual function:

```
class X {
public:
  friend ostream& operator<< (ostream& o,const X& x)
    { x.print(o); return o; }
  //...
protected:
  virtual void print(ostream& o) const;  //or `=0;' if `X' is abstract
};
```

Now all subclasses of X merely provide their own `print(ostream&)const' member function, and they all share the common `<<' operator. Friends don't bind dynamically, but this technique makes them *act* as if they were.

Q75: What is a `virtual destructor'?

A: In general, a virtual fn means to start at the class of the object itself, not the type of the pointer/ref (`do the right thing based on the actual class of' is a good way to remember it). Virtual destructors (dtors) are no different: start the destruction process `down' at the object's actual class, rather than `up' at the ptr's class (ie: `destroy yourself using the *correct* destruction routine').

Virtual destructors are so valuable that some people want compilers to holler at you if you forget them. In general there's only one reason *not* to make a class' dtor virtual: if that class has no virtual fns, the introduction of the first virtual fn imposes typically 4 bytes overhead in the size of each object (there's a bit of magic for how C++ `does the right thing', and it boils down to an extra ptr per object called the `virtual table pointer' or `vptr').

Q76: What is a `virtual constructor'?

A: Technically speaking, there is no such thing. You can get the effect you desire by a virtual `create_copy()' member fn (for copy constructing), or a `create_similar()' member fn (also virtual) which constructs/creates a new object of the same class but is `fresh' (like the `default' [zero parameter] ctor would do).

The reason ctors can't be virtual is simple: a ctor turns raw bits into a living object. Until there's a living object against which to invoke the member function, you can't expect a member function invocation to be handled `the right way'. You can think of ctors as `class' [static] functions, or as `factories' which churn out objects. Thinking of ctors as `methods' attached to an object is misleading.

Here is an example of how you could use `create_copy()' and `create_similar()' methods:

```
class Set {             //normally this would be a template
public:
  virtual void insert(int);   //Set of `int'
  virtual int  remove();
  //...
  virtual Set* create_copy() const = 0;   //pure virtual; Set is an ABC
  virtual Set* create_similar() const = 0;
  virtual ~Set() { }          //see on `virtual destructors' for more
};
class SetHT : public Set {
public:
  //...
  Set* create_copy()    const { return new SetHT(*this); }
  Set* create_similar() const { return new SetHT(); }
protected:
  //a hash table in here
};
```

A SetHT is-a Set, so the return value is correct. The invocation of `SetHT(*this)' is that of copy construction (`*this' has type `const SetHT&'). Although `create_copy()' returns a new SetHT, the caller of create_copy() merely knows he has a Set, not a SetHT (which is desirable in the case of wanting a `virtual ctor'). `create_similar()' is similar, but it constructs an `empty' SetHT.

Clients can use this as if they were `virtual constructors':

```
void client_code(Set& s)
{
  Set* s2 = s.create_copy();
  Set* s3 = s.create_similar();
  //...
  delete s2;      //relies on destructor being virtual!!
  delete s3;      // ditto
}
```

This fn will work correctly regardless of how the Set is implemented (hash table based, AVL tree based, etc).

See above on `separation of interface from implementation' for more.

Q77: What are some good C++ coding standards?

A: Thank you for reading this answer rather than just trying to set your own coding standards. But please don't ask this question on Usenet. Nearly every software engineer has, at some point, felt that coding standards are or can be used as a `power play'. Furthermore some attempts to set C++ coding standards have been made by those unfamiliar with the language and/or paradigm, so the standards end up being based on what *was* the state-of-the-art when the setters where writing code. Such impositions generate an attitude of mistrust for coding standards. Obviously anyone who asks this question on Usenet wants to be trained so they *don't* run off on their own ignorance, but nonetheless the answers tend to generate more heat than light.

Q78: Are coding standards necessary? sufficient?

A: Coding standards do not make non OO programmers into OO programmers. Only training and experience do that. If they have merit, it is that coding standards discourage the petty fragmentation that occurs when organizations coordinate the activities of diverse groups of programmers.

But you really want more than a coding standard. The structure provided by coding standards gives neophytes one less degree of freedom to worry about, however pragmatics go well beyond pretty-printing standards. We actually need a consistent *philosophy* of implementation. Ex: strong or weak typing? references or ptrs in our interface? stream I/O or stdio? should C++ code call our C? vise versa? should we use ABCs? polymorphism? inheritance? classes? encapsulation? how should we handle exceptions? etc.

Therefore what is needed is a `pseudo standard' for detailed *design*. How can we get this? I recommend a two-pronged approach: training and libraries. Training provides `intense instruction', and a high quality C++ class library provides `long term instruction'. There is a thriving commercial market for both kinds of `training'. Advice by organizations who have been through the mill is consistent: Buy, Don't Build. Buy libraries, buy training, buy tools. Companies who have attempted to become a self-taught tool-shop as well as an application/system shop have found success difficult.

Few argue that coding standards are `ideal', or even `good', however many feel that they're necessary in the kind of organizations/situations described above.

The following questions provide some basic guidance in conventions and styles.

Q79: Should our organization determine coding standards from our C experience?

A: No matter how vast your C experience, no matter how advanced your C expertise, being a good C programmer does not make you a good C++ programmer. C programmers must learn to use the `++' part of `C++', or the results will be lackluster. People who want the `promise' of OOP, but who fail to put the `OO' into OOP, are fooling themselves, and the balance sheet will show their folly.

C++ coding standards should be tempered by C++ experts. Asking comp.lang.c++ is a start (but don't use the term `coding standard' in the question; instead simply say, `what are the pros and cons of this technique?'). Seek out experts who can help guide you away from pitfalls. Get training. Buy libraries and see if `good' libraries pass your coding standards. Do *not* set standards by yourself unless you have considerable experience in C++. Having no standard is better than having a bad standard, since improper `official' positions `harden' bad brain traces. There is a thriving market for both C++ training and libraries from which to pool expertise.

One more thing: whenever something is in demand, the potential for charlatans increases. Look before you leap. Also ask for student-reviews from past companies, since not even expertise makes someone a good communicator. Finally, select a practitioner who can teach, not a full time teacher who has a passing knowledge of the language/paradigm.

Q80: Should I declare locals in the middle of a fn or at the top?

A: Different people have different opinions about coding standards. However one thing we all should agree on is this: no style guide should impose undue performance penalties. The real reason C++ allows objects to be created anywhere in the block is not style, but performance.

An object is initialized (constructed) the moment it is declared. If you don't have enough information to initialize an object until half way down the fn, you can either initialize it to an `empty' value at the top then `assign' it later, or initialize it correctly half way down the fn. It doesn't take much imagination to see that it's cheaper to get it right the first time than it is to build it once, tear it down, then rebuild it again. Simple examples show a factor of 350% speed hit for simple classes like String. Your mileage may vary; surely the overall system degradation will be less that 300+%, but there *will* be degradation. *Unnecessary* degradation.

A common retort to the above is: `we'll provide "set" methods for every datum in our objects, so the cost of construction will be spread out'. This is worse than the performance overhead, since now you're introducing a maintenance nightmare. Providing `set' methods for every datum is tantamount to public data. You've exposed your implementation technique to the world. The only thing you've hidden is the physical *names* of your subobjects, but the fact that you're using a List and a String and a float (for example) is open for all to see. Maintenance generally consumes far more resources than run-time CPU.

Conclusion: in general, locals should be declared near their first use. Sorry that this isn't `familiar' to your C experts, but `new' doesn't necessarily mean `bad'.

Q81: What source-file-name convention is best? `foo.C'? `foo.cc'? `foo.cpp'?

A: Most Un*x compilers accept `.C' for C++ source files, g++ preferring `.cc', and cfront also accepting `.c'. Most DOS and OS/2 compilers require `.cpp' since DOS filesystems aren't case sensitive. Some also advocate `.cxx'. The impact of this decision is not great, since a trivial shell script can rename all .cc files into .C files. The only files that would have to be modified are the Makefiles, which is a very small piece of your maintenance costs. Note however that some versions of cfront accept a limited set of suffixes (ie: some can't handle `.cc'; in these cases it is easier to tell `make' about CC's convention than vise versa).

You can use `.C' on DOS or OS/2 if the compiler provides a command-line option to tell it to always compile with C++ rules (ex: `ztc -cpp foo.C' for Zortech, `bcc -P foo.C' for Borland, etc).

Q82: What header-file-name convention is best? `foo.H'? `foo.hh'? `foo.hpp'?

A: The naming of your source files is cheap since it doesn't affect your source code. Your substantial investment is your source code. Therefore the names of your header files must be chosen with much greater care. The preprocessor will accept whatever name you give it in the #include line, but whatever you choose, you will want to plan on sticking with it for a long time, since it is more expensive to change (though certainly not as difficult as, say, porting to a new language).

Almost all vendors ship their C++ header files using a `.h' extension, which means you can reliably do things like:

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

Some sites use `.H' for their own internally developed header files, but most simply use `.h'.

Q83: Are there any lint-like guidelines for C++?

A: Yes, there are some practices which are generally considered dangerous. However none of these are universally `bad', since situations arise when even the worst of these is needed:

* a class `X's assignment operator should return `*this' as an `X&' (allows chaining of assignments)
* a class with any virtual fns ought to have a virtual destructor
* a class with any of {dtor, assignment-op, copy-ctor} generally needs all 3
* a class `X's copy-ctor and assignment-op should have `const' in the param:
  `X::X(const X&)' and `X& X::operator=(const X&)' respectively
* always use initialization lists for class sub-objects rather than assignment
  the performance difference for user-defined classes can be substantial (3x!)
* many assignment operators should start by testing if `we' are `them'; ex:

```
X& X::operator=(const X& x)
{
  if (this == &x) return *this;
  //...normal assignment duties...
  return *this;
}
```

  sometimes there is no need to check, but these situations generally correspond to when there's no need for an explicit user-specified assignment op (as opposed to a compiler-synthesized assignment-op).

* in classes that define both `+=', `+' and `=', `a+=b' and `a=a+b' should generally do the same thing; ditto for the other identities of builtin types (ex: a+=1 and ++a; p[i] and *(p+i); etc). This can be enforced by writing the binary ops using the `op=' forms; ex:

```
X operator+(const X& a, const X& b)
{
  X ans = a;
  ans += b;
  return ans;
}
```

This way the `constructive' binary ops don't even need to be friends. But it is sometimes possible to more efficiently implement common ops (ex: if class `X' is actually `String', and `+=' has to reallocate/copy string memory, it may be better to know the eventual length from the beginning).

Q84: Why does C++'s FAQ have a section on Smalltalk? Is this Smalltalk-bashing?

A: The two `major' OOPLs in the world are C++ and Smalltalk. Due to its popularity as the OOPL with the second largest user pool, many new C++ programmers come from a Smalltalk background. This section answers the questions:

* what's different about the two languages
* what must a Smalltalk-turned-C++ programmer know to master C++

This section does *!*NOT*!* attempt to answer the questions:
* which language is `better'?
* why is Smalltalk `bad'?

Nor is it an open invitation for some Smalltalk terrorist to slash my tires while I sleep (on those rare occasions when I have time to rest these days :-).

Q85: What's the difference between C++ and Smalltalk?

A: There are many differences such as compiled vs perceived-as-interpreted, pure vs hybrid, faster vs perceived-as-slower, etc. Some of these aren't true (ex: a large portion of a typical Smalltalk program can be compiled by current implementations, and some Smalltalk implementations perform reasonably well). But none of these affect the programmer as much as the following three issues:

* static typing vs dynamic typing (`strong' and `weak' are synonyms)
* how you use inheritance
* value vs reference semantics

The first two differences are illuminated in the remainder of this section; the third point is the subject of the section that follows.

If you're a Smalltalk programmer who wants to learn C++, you'd be very wise to study the next three questions carefully. Historically there have been many attempts to `make' C++ look/act like Smalltalk, even though the languages are very Very different. This hasn't always led to failures, but the differences are significant enough that it has led to a lot of needless frustration and expense. The quotable quote of the year goes to Bjarne Stroustrup at the `C++ 1995' panel discussion, 1990 C++-At-Work conference, discussing library design:

`Smalltalk is the best Smalltalk around'.

Q86: What is `static typing', and how is it similar/dissimilar to Smalltalk?

A: Static (most say `strong') typing says the compiler checks the type-safety of every operation *statically* (at compile-time), rather than to generate code which will check things at run-time. For example, the signature matching of fn arguments is checked, and an improper match is flagged as an error by the *compiler*, not at run-time.

In OO code, the most common `typing mismatch' is invoking a member function against an object which isn't prepared to handle the operation. Ex: if class `X' has member fn f() but not g(), and `x' is an instance of class X, then x.f() is legal and x.g() is illegal. C++ (statically/strongly typed) catches the error at compile time, and Smalltalk (dynamically/weakly typed) catches `type' errors at run-time. (Technically speaking, C++ is like Pascal [*pseudo* statically typed], since ptr casts and unions can be used to violate the typing system; you probably shouldn't use these constructs very much).

Q87: Which is a better fit for C++: `static typing' or `dynamic typing'?

A: The arguments over the relative goodness of static vs dynamic typing will continue forever. However one thing is clear: you should use a tool like it was intended and designed to be used. If you want to use C++ most effectively, use it as a statically typed language. C++ is flexible enough that you can (via ptr casts, unions, and #defines) make it `look' like Smalltalk.

There are places where ptr casts and unions are necessary and even wholesome, but they should be used carefully and sparingly. A ptr cast tells the compiler to believe you. It effectively suspends the normal type checking facilities. An incorrect ptr cast might corrupt your heap, scribble into memory owned by other objects, call nonexistent methods, and cause general failures. It's not a pretty sight. If you avoid these and related constructs, you can make your C++ code both safer and faster -- anything that can be checked at compile time is something that doesn't have to be done at run-time, one `pro' of strong typing.

Even if you're in love with weak typing, please consider using C++ as a strongly typed OOPL, or else please consider using another language that better supports your desire to defer typing decisions to run-time. Since C++ performs 100% type checking decisions at compile time, there is *no* built-in mechanism to do *any* type checking at run-time; if you use C++ as a weakly typed OOPL, you put your life in your own hands.

Q88: How can you tell if you have a dynamically typed C++ class library?

A: One hint that a C++ class library is weakly typed is when everything is derived from a single root class, usually `Object'. Even more telling is the implementation of the container classes (List, Stack, Set, etc): if these containers are non-templates, and if their elements are inserted/extracted as ptrs to `Object', the container will promote weak typing. You can put an Apple into such a container, but when you get it out, the compiler only knows that it is derived from Object, so you have to do a pointer cast (a `down cast') to cast it `down' to an Apple (you also might hope a lot that you got it right, cause your blood is on your own head).

You can make the down cast `safe' by putting a virtual fn into Object such as `are_you_an_Apple()' or perhaps `give_me_the_name_of_your_class()', but this dynamic testing is just that: dynamic. This coding style is the essence of weak typing in C++. You call a function that says `convert this Object into an Apple or kill yourself if its not an Apple', and you've got weak typing: you don't know if the call will succeed until run-time.

When used with templates, the C++ compiler can statically validate 99% of an application's typing information (the figure `99%' is apocryphal; some claim they always get 100%, others find the need to do persistence which cannot be statically type checked). The point is: C++ gets genericity from templates, not from inheritance.

Q89: Will `standard C++' include any dynamic typing primitives?

A: Yep.

Note that the effect of a down-cast and a virtual fn call are similar: in the member fn that results from the virtual fn call, the `this' ptr is a downcasted version of what it used to be (it went from ptr-to-Base to ptr-to-Derived). The difference is that the virtual fn call *always* works: it never makes the wrong `down-cast' and it automatically extends itself whenever a new subclass is created -- as if an extra `case' or `if/else' magically appearing in the weak typing technique. The other difference is that the client gives control to the object rather than reasoning *about* the object.

Q90: How do you use inheritance in C++, and is that different from Smalltalk?

A: There are two reasons one might want to use inheritance: to share code, or to express your interface compliance. Ie: given a class `B' (`B' stands for `base class', which is called `superclass' in Smalltalkese), a class `D' which is derived from B is expressed this way:

```
class B { /*...*/ };
class D : public B { /*...*/ };
```

This says two distinct things: (1) the bits(data structure) + code(algorithms) are inherited from B, and (2) `D's public interface is `conformal' to `B's (anything you can do to a B, you can also do to a D, plus perhaps some other things that only D's can do; ie: a D is-a-kind-of-a B).

In C++, one can use inheritance to mean:
--> #2(is-a) alone (ex:you intend to override most/all inherited code)
--> both #2(is-a) and #1(code-sharing)
but one should never Never use the above form of inheritance to mean
--> #1(code-sharing) alone (ex: D really *isn't* a B, but...)

This is a major difference with Smalltalk, where there is only one form of inheritance (C++ provides `private' inheritance to mean `share the code but don't conform to the interface'). The Smalltalk language proper (as opposed to coding practice) allows you to have the *effect* of `hiding' an inherited method by providing an override that calls the `does not understand' method. Furthermore Smalltalk allows a conceptual `is-a' relationship to exist *apart* from the subclassing hierarchy (subtypes don't have to be subclasses; ex: you can make something that `is-a Stack' yet doesn't inherit from `Stack').

In contrast, C++ is more restrictive about inheritance: there's no way to make a `conceptual is-a' relationship without using inheritance (the C++ work-around is to separate interface from implementation via ABCs). The C++ compiler exploits the added semantic information associated with public inheritance to provide static typing.

Q91: What are the practical consequences of diffs in Smalltalk/C++ inheritance?

A: Since Smalltalk lets you make a subtype without making a subclass, one can be very carefree in putting data (bits, representation, data structure) into a class (ex: you might put a linked list into a Stack class). After all, if someone wants something that an array-based-Stack, they don't have to inherit from Stack; they can go off and make effectively a stand-alone class (they might even *inherit* from an Array class, even though they're not-a-kind-of- Array!).

In C++, you can't be nearly as carefree. Since only mechanism (method code), but not representation (data bits) can be overridden in subclasses, you're usually better off *not* putting the data structure in a class. This leads to the concept of Abstract Base Classes (ABCs), which are discussed in a separate question. You can change the algorithm but NOT the data structure. Bits are forever.

I like to think of the difference between an ATV and a Maseratti. An ATV [all terrain vehicle] is more fun, since you can `play around' by driving through fields, streams, sidewalks and the like. A Maseratti, on the other hand, gets you there faster, but it forces you to stay on the road. My advice to C++ programmers is simple: stay on the road. Even if you're one of those people who like the `expressive freedom' to drive through the bushes, don't do it in C++; it's not a good `fit'.

Note that C++ compilers uphold the is-a semantic constraint only with `public' inheritance. Neither containment (has-a), nor private or protected inheritance implies conformance.

Q92: Do you need to learn a `pure' OOPL before you learn C++?

A: The short answer is, No.

The medium answer length answer is: learning some `pure' OOPLs may *hurt* rather than help.

The long answer is: read the previous questions on the difference between C++ and Smalltalk (the usual `pure' OOPL being discussed; `pure' means everything is an object of some class; `hybrid' [like C++] means things like int, char, and float are not instances of a class, hence aren't subclassable).

The `purity' of the OOPL doesn't make the transition to C++ any more or less difficult; it is the weak typing and improper inheritance that is so hard to get. I've taught numerous people C++ with a Smalltalk background, and they usually have just as hard a time as those who've never seen inheritance before. In fact, my personal observation is that those with extensive experience with a weakly typed OOPL (usually but not always Smalltalk) have a *harder* time, since it's harder to *unlearn* habits than it is to learn the statically typed way from the beginning.

Q93: What is the NIHCL? Where can I get it?

A: NIHCL stands for `national-institute-of-health's-class-library'. it can be acquired via anonymous ftp from [128.231.128.7] in the file pub/nihcl-3.0.tar.Z

NIHCL (some people pronounce it `N-I-H-C-L', others pronounce it like `nickel') is a C++ translation of the Smalltalk class library. There are some ways where NIHCL's use of weak typing helps (ex: persistent objects). There are also places where the weak typing it introduces create tension with the underlying statically typed language.

A draft version of the 250pp reference manual is included with version 3.10 (gnu emacs TeX-info format). It is not available via uucp, or via regular mail on tape, disk, paper, etc (at least not from Keith Gorlen).

See previous questions on Smalltalk for more.

Q94: What is value and/or reference semantics, and which is best in C++?

A: With reference semantics, assignment is a pointer-copy (ie: a *reference*). Value (or `copy') semantics mean assignment copies the value, not just the pointer. C++ gives you the choice: use the assignment operator to copy the value (copy/value semantics), or use a ptr-copy to copy a pointer (reference semantics). C++ allows you to override the assignment operator to do anything your heart desires, however the default (and most common) choice is to copy the *value*. Smalltalk and Eiffel and CLOS and most other OOPLs force reference semantics; you must use an alternate syntax to copy the value (clone, shallowCopy, deepCopy, etc), but even then, these languages ensure that any name of an object is actually a *pointer* to that object (Eiffel's `expanded' classes allow a supplier-side work-around).

There are many pros to reference semantics, including flexibility and dynamic binding (you get dynamic binding in C++ only when you pass by ptr or pass by ref, not when you pass by value).

There are also many pros to value semantics, including speed. `Speed' seems like an odd benefit to for a feature that requires an object (vs a ptr) to be copied, but the fact of the matter is that one usually accesses an object more than one copies the object, so the cost of the occasional copies is (usually) more than offset by the benefit of having an actual object rather than a ptr to an object.

There are three cases when you have an actual object as opposed to a pointer to an object: local vars, global/static vars, and fully contained subobjects in a class. The most common & most important of these is the last (`containment').

More info about copy-vs-reference semantics is given in the next questions. Please read them all to get a balanced perspective. The first few have intentionally been slanted toward value semantics, so if you only read the first few of the following questions, you'll get a warped perspective.

Assignment has other issues (ex: shallow vs deep copy) which are not covered here.

Q95: What is `virtual data', and how-can / why-would I use it in C++?

A: Virtual data isn't strictly a `part' of C++, however it can be simulated. It's not entirely pretty, but it works. First we'll cover what it is and how to simulate it, then conclude with why it isn't `part' of C++.

Consider classes Vec (like an array of int) and SVec (a stretchable Vec; ie: SVec overrides operator[] to automatically stretch the number of elements whenever a large index is encountered). SVec inherits from Vec. Naturally Vec's subscript operator is virtual.

Now consider a VStack class (Vec-based-Stack). Naturally this Stack has a capacity limited by the fixed number of elements in the underlying Vec data structure. Then someone comes along and wants an SVStack class (SVec based Stack). For some reason, they don't want to merely modify VStack (say, because there are many users already using it).

The obvious choice then would be to inherit SVStack from VStack, however then there'd be *two* Vecs in an SVStack object (one explicitly in VStack, the other as the base class subobject in the SVec which is explicitly in the SVStack). That's a lot of extra baggage. There are at least 2 solns:

* break the is-a link between SVStack and VStack, text-copy the code from VStack and manually change `Vec' to `SVec'.
* activate some sort of virtual data, so subclasses can change the class of subobjects.

To effect virtual data, we need to change the Vec subobject from a physically contained subobject into a ptr pointing to a dynamically allocated subobject:

```
_____original_____                |_____to_support_virtual_data
class VStack {                    | class VStack {
public:                           | public:
  VStack(int cap=10)              |   VStack(int cap=10)
    : v(cap), sp(0) { }           |     : v(*new Vec(cap)), sp(0) { }
//FREESTORE
  void push(int x) {v[sp++]=x;} |   void push(int x) {v[sp++]=x;}   //no
change
  int  pop()  {return v[--sp];} |   int  pop()  {return v[--sp];}   //no
change
 ~VStack() { }   //unnecessary  |  ~VStack()    {delete &v;}
//NECESSARY
protected:                        | protected:
  Vec v;  //where data stored    |   Vec& v; //where data is stored
  int sp; //stack pointer        |   int sp; //stack pointer
};                                | };
```

Now the subclass has a shot at overriding the defn of the object referred to as `v'. Ex: basically SVStack merely needs to bind a new SVec to `v', rather than letting VStack bind the Vec. However classes can only initialize their *own* subobjects in an init-list. Even if I had used a ptr rather than a ref, VStack must be prevented from allocating its own `Vec'. The way we do this is to add another ctor to VStack that takes a Vec& and does *not* allocate a Vec:

```
class VStack {
protected:
  VStack(Vec& vv) : v(vv), sp(0) { }      //`protected' constructor!
//...                          //(prevents public access)
};
```

That's all there is to it! Now the subclass (SVStack) can be defined as:

```
class SVStack : public VStack {
public:
  SVStack(int init_cap=10) : VStack(*new SVec(init_cap)) { }
};
```

Pros: * implementation of SVStack is a one-liner
* SVStack shares code with VStack

Cons: * extra layer of indirection to access the Vec
* extra freestore allocations (both new and delete)
* extra dynamic binding (reason given in next question)

We succeeded at making *our* job easier as implementor of SVStack, but all clients pay for it. It wouldn't be so bad if clients of SVStack paid for it, after all, they chose to use SVStack (you pay for it if you use it). However the `optimization' made the users of the plain VStack pay as well!

See the question after the next to find out how much the client's `pay'. Also: *PLEASE* read the few questions that follow the next one too (YOU WILL NOT GET A BALANCED PERSPECTIVE WITHOUT THE OTHERS).

Q96: What's the difference between virtual data and dynamic data?

A: The easiest way to see the distinction is by an analogy with `virtual fns': A virtual member fn means the declaration (signature) must stay the same in subclasses, but the defn (body) can be overridden. The overriddenness of an inherited member fn is a static property of the subclass; it doesn't change dynamically throughout the life of any particular object, nor is it possible for distinct objects of the subclass to have distinct defns of the member fn.

Now go back and re-read the previous paragraph, but make these substitutions:

```
`member fn' --> `subobject'
`signature' --> `type'
`body'      --> `exact class'
```

After this, you'll have a working defn of virtual data. `Per-object member fns' (a member fn `f()' which is potentially different in any given instance of an object) could be handled by burying a function ptr in the object, then setting the (const) fn ptr during construction.

`Dynamic member fns' (member fns which change dynamically over time) could also be handled by function ptrs, but this time the fn ptr would not be const. In the same way, there are three distinct concepts for data members:

* virtual data: the defn (`class') of the subobject is overridable in subclasses provided its declaration (`type') remains the same, and this overriddenness is a static property of the [sub]class.
* per-object-data: any given object of a class can instantiate a different conformal (same type) subobject upon initialization (usually a `wrapper' object), and the exact class of the subobject is a static property of the object that wraps it.
* dynamic-data: the subobject's exact class can change dynamically over time.

The reason they all look so much the same is that none of this is `supported' in C++. It's all merely `allowed', and in this case, the mechanism for faking each of these is the same: a ptr to a (probably abstract) base class. In a language that made these `first class' abstraction mechanisms, the difference would be more striking, since they'd each have a different syntactic variant.

Q97: Should class subobjects be ptrs to freestore allocated objs, or contained?

A: Usually your subobjects should actually be `contained' in the aggregate class (but not always; `wrapper' objects are a good example of where you want a a ptr/ref; also the N-to-1-uses-a relationship needs something like a ptr/ref).

There are three reasons why fully contained subobjects have better performance than ptrs to freestore allocated subobjects:

* extra layer to indirection every time you need to access subobject
* extra freestore allocations (`new' in ctor, `delete' in dtor)
* extra dynamic binding (reason given later in this question)

Q98: What are relative costs of the 3 performance hits of allocated subobjects?

A: The three performance hits are enumerated in the previous question:

* By itself, an extra layer of indirection is small potatoes.
* Freestore allocations can be a big problem (standard malloc's performance degrades with more small freestore allocations; OO s/w can easily become `freestore bound' unless you're careful).
* Extra dynamic binding comes from having a ptr rather than an object. Whenever the C++ compiler can know an object's *exact* class, virtual fn calls can be *statically* bound, which allows inlining. Inlining allows zillions (would you believe half a dozen :-) optimization opportunities such as procedural integration, register lifetime issues, etc. The C++ compiler can know an object's exact class in three circumstances: local variables, global/static variables, and fully-contained subobjects.

Thus fully-contained subobjects allow significant optimizations that wouldn't be possible under the `subobjects-by-ptr' approach (this is the main reason that languages which enforce reference-semantics have `inherent' performance problems).

Q99: What is an `inline virtual member fn'? Are they ever actually `inlined'?

A: A inline virtual member fn is a member fn that is inline and virtual :-). The second question is much harder to answer. The short answer is `Yes, but'.

A virtual call (msg dispatch) via a ptr or ref is always resolved dynamically (at run-time). In these situations, the call is never inlined, since the actual code may be from a derived class that was created after the caller was compiled.

The difference between a regular fn call and a virtual fn call is rather small. In C++, the cost of dispatching is rarely a problem. But the lack of inlining in any language can be very Very significant. Ex: simple experiments will show the difference to get as bad as an order of magnitude (for zillions of calls to insignificant member fns, loss of inlining virtual fns can result in 25X speed degradation! [Doug Lea, `Customization in C++', proc Usenix C++ 1990]).

This is why endless debates over the actual number of clock cycles required to do a virtual call in language/compiler X on machine Y are largely meaningless. Ie: many language implementation vendors make a big stink about how good their msg dispatch strategy is, but if these implementations don't *inline* method calls, the overall system performance would be poor, since it is inlining --*not* dispatching-- that has the greatest performance impact.

NOTE: PLEASE READ THE NEXT TWO QUESTIONS TO SEE THE OTHER SIDE OF THIS COIN!

Q100: Sounds like I should never use reference semantics, right?

A: Wrong.

Reference semantics is A Good Thing. We can't live without pointers. We just don't want our s/w to be One Gigantic Pointer. In C++, you can pick and choose where you want reference semantics (ptrs/refs) and where you'd like value semantics (where objects physically contain other objects etc). In a large system, there should be a balance. However if you implement absolutely *everything* as a pointer, you'll get enormous speed hits.

Objects near the problem skin are larger than higher level objects. The *identity* of these `problem space' abstractions is usually more important than their `value'. These combine to indicate reference semantics should be used for problem-space objects (Booch says `Entity Abstractions'; see on `Books').

The question arises: is reference semantics likely to cause a performance problem in these `entity abstractions'? The key insight in answering this question is that the relative interaction frequency is much lower for problem skin abstractions than for low level server objects.

Thus we have an *ideal* situation in C++: we can choose reference semantics for objects that need unique identity or that are too large to copy, and we can choose value semantics for the others. The result is very likely to be that the highest frequency objects will end up with value semantics. Thus we install flexibility only where it doesn't hurt us, and performance where we need it most!

These are some of the many issues the come into play with real OO design. OO/C++ mastery takes time and high quality training. That's the investment-price you pay for a powerful tool.

<<<<DON'T STOP NOW! READ THE NEXT QUESTION TOO!!>>>>

Q101: Does the poor performance of ref semantics mean I should pass-by-value?

A: No. In fact, `NO!' :-)

The previous questions were talking about ***subobjects***, not parameters. Pass- by-value is usually a bad idea when mixed with inheritance (larger subclass objects get `sliced' when passed by value as a base class object). Generally, objects that are part of an inheritance hierarchy should be passed by ref or by ptr, but not by value, since only then do you get the (desired) dynamic binding.

Unless compelling reasons are given to the contrary, subobjects should be by value and parameters should be by reference. The discussion in the previous few questions indicates some of the `compelling reasons' for when subobjects should be by reference.

Q102: How can I call a C function `f()' from C++ code?

A: Tell the C++ compiler that it is a C function: extern "C" void f(); Be sure to include the full function prototype. A block of many C functions can be grouped via braces, as in:

```
extern "C" {
  void* malloc(size_t);
  char* strcpy(char* dest, const char* src);
  int   printf(const char* fmt, ...);
}
```

Q103: How can I create a C++ function `f()' that is callable by my C code?

A: Use the same `extern "C" f()' construct as detailed in the previous question, only then proceed to actually define the function in your C++ module. The compiler will ensure that the external information sent to the linker uses C calling conventions and name mangling (ex: preceded by a single underscore). Obviously you can't make several overloaded fns simultaneously callable by a C program, since name overloading isn't supported by C.

Caveats and implementation dependencies:
* your `main()' should be compiled with your C++ compiler.
* your C++ compiler should direct the linking process.

Q104: Why's the linker giving errors for C/C++ fns being called from C++/C fns?

A: See the previous two questions on how to use `extern "C"'.

Q105: How can I pass an object of a C++ class to/from a C function?

A: Here's an example of one that will work (be sure to read the tail of this answer which details when such a scheme will ***not*** work):

```
/****** C/C++ header file: X.h ******/
#ifdef __cplusplus    /*`__cplusplus' is #defined iff compiler is C++*/
  extern "C" {
#endif

#ifdef __STDC
  extern int c_fn(struct X*);       /* ANSI-C prototypes */
  extern struct X* cplusplus_callback_fn(struct X*);
#else
  extern int c_fn();                /* K&R style */
  extern struct X* cplusplus_callback_fn();
#endif

#ifdef __cplusplus
  }
#endif

#ifdef __cplusplus
  class X {
    int a;
  public:
    X();
    void frob(int);
  };
#endif
```

Then, in file `X.C':

```
#include "X.h"
X::X() : a(0) { }
void X::frob(int aa) { a = aa; }

X* cplusplus_callback_fn(X* x)
{
  x->frob(123);
  return x;
}
```

In C++ file `main.C':

```
#include "X.h"

int main()
{
  X x;
  c_fn(&x);
  return 0;
}
```

Finally, in a C file `c-fn.c':

```
#include "X.h"

int c_fn(struct X* x)
{
  if (cplusplus_callback_fn(x))
    do_one_thing();
```

```
    else
      do_something_else();
    return something();
  }
```

Passing ptrs to C++ objects to/from C fns will FAIL if you pass and get back something that isn't *exactly* the same pointer, such as passing a base class ptr and receiving a derived class ptr (this fails when multiple inheritance is involved, since C fails to do pointer-conversion properly).

Q106: Can my C function access data in an object of a C++ class?

A: Sometimes.

(First read the previous question on passing C++ objects to/from C functions.) You can safely access a C++ object's data from a C function if the C++ class:

* has no virtual functions (including inherited virtual fns)

* has all its data in the same access-level section (private/protected/public)

* has no fully-contained subobjects with virtual fns

If the C++ class has any base classes at all (or if any fully contained subobjects have base classes), accessing the data will *technically* be non-portable, since class layout under inheritance isn't imposed by the language. However in practice, all C++ compilers do it the same way: the base class object appears first (in left-to-right order in the event of multiple inheritance), and subobjects follow.

Furthermore you can often (but less than always) assume a `void*' appears in the object at the location of the first virtual function. This is trickier, since the first virtual function is often in a different access specifier section than the data members. Even the use of a single pointer is not required by the language (but this is the way `everyone' does it).

If the class has any virtual base classes, it is more complicated and less portable. One common implementation technique is for objects to contain an object of the virtual base class (V) last (regardless of where `V' shows up as a virtual base class in the inheritance DAG), with the rest of the object's parts appearing in the normal order. Every class that has V as a virtual base class actually has a *pointer* to the V part of the final object.

Q107: Why do I feel like I'm `further from the machine' in C++ as opposed to C?

A: Because you are. Being an OOPL, C++ allows you to directly model the problem domain itself (obviously OOD is more complex than this, but a good start at `finding the objects' is to model the nouns in the problem domain). By modeling the problem domain, the system interacts in the language of the problem domain rather than in the language of the solution domain.

One of C's great strengths is the fact that it has `no hidden mechanism'. What you see is what you get. You can read a C program and `see' every clock cycle. This is not the case in C++; overloaded operators are a case in point. Old line C programmers (such as many of us once were) are often ambivalent about this feature, but soon we realize that it provides a level of abstraction and economy of expression which lowers maintenance costs without destroying runtime performance.

Naturally you can write assembly code in any language; using C++ doesn't guarantee any particular level of quality, reusability, abstraction, or any other measure of `goodness'. C++ doesn't try to make it impossible for bad programmers to write bad programs; it enables good programmers to write good programs.

Q108: What is the type of `ptr-to-member-fn'?  Is it diffn't from `ptr-to-fn'?

A: A member fn of class X has type: Returntype (X::*)(Argtypes)
   while a plain function has type: Returntype (*) (Argtypes)

Q109: How can I ensure `X's objects are only created with new, not on the stack?

A: Make constructors protected and define `friend' or `static' fns that return a ptr to objects created via `new' (the ctors must be protected rather than private, otherwise you couldn't derive from the class). Ex:

```
class X {   //only want to allow dynamicly allocated X's
public:
  static X* create()           { return new X();  }
  static X* create(int i)      { return new X(i); }
  static X* create(const X& x) { return new X(x); }
protected:
           X();
           X(int i);
           X(const X& x);
  virtual ~X();
};
X* Xptr = X::create(5);
```

Q110: How do I pass a ptr to member fn to a signal handler,X event callback,etc?

A: Because a member function is meaningless without an object to invoke it on, you can't do this directly (if `X' were rewritten in C++, it would probably pass references to *objects* around, not just pointers to fns; naturally the objects would embody the required function and probably a whole lot more). As a patch for existing software, use a free function as a wrapper which takes an object obtained through some other technique (held in a global, perhaps) and calls the desired member function. There is one exception: static member functions do not require an actual object to be invoked, and ptrs-to-static- member-fns are type compatible with regular ptrs-to-fns (see ARM p.25, 158).

Ex: suppose you want to call X::memfn() on interrupt:

```
    class X {
    public:
              void memfn();
      static void staticmemfn();   //a static member fn can handle it
      //...
    };

    //wrapper fn remembers the object on which to invoke memfn in a static
var:
    static X* object_which_will_handle_signal;
    void X_memfn_wrapper() { object_which_will_handle_signal.memfn(); }

    main()
    {
      /* signal(SIGINT, X::memfn); */   //Can NOT do this
      signal(SIGINT, X_memfn_wrapper);  //Ok
      signal(SIGINT, X::staticmemfn);   //Also Ok
    }
```

Q111: Why am I having trouble taking the address of a C++ function?

Short ans: Please read previous question first; this is a corollary.

Long ans: In C++, member fns have an implicit parameter which points to the object (the `this' ptr inside the member fn). Normal C fns can be thought of as having a different calling convention from member fns, so the types of their ptrs (ptr-to-member-fn vs ptr-to-fn) are different and incompatible. C++ introduces a new type of ptr, called a ptr-to-member, which can only be invoked by providing an object (see ARM 5.5). Do NOT attempt to `cast' a ptr-to-mem-fn into a ptr-to-fn; the result is undefined and probably disastrous; a ptr-to- member-fn is NOT required to contain the machine addr of the appropriate fn (see ARM, 8.1.2c, p.158). As was said in the last example, if you want a regular C fn ptr, use either a top-level (non-class) fn, or a `static' (class) member fn.

Q112: How do I declare an array of pointers to member functions?

A: Use the following declaration:

```
class Frob {
public:
  Rettype f(T1 x, T2 y);
  Rettype g(T1 x, T2 y);
  Rettype h(T1 x, T2 y);
  Rettype i(T1 x, T2 y);
  //...
};

Rettype (Frob::*fn_ptr[3])(T1,T2) = { &Frob::f, &Frob::g, &Frob::h };
```

You can make the array declaration somewhat clearer with a typedef:

```
typedef  Rettype (Frob::*Frob_member_ptr)(T1,T2);
//...
Frob_member_ptr fn_ptr[3] = { &Frob::f, &Frob::g, &Frob::h };
```

To call one of the functions on an object `frob', use:

```
Frob frob;
//...
(frob.*fn_ptr[i])(x, y);
```

You can make the call somewhat clearer using a #define:

```
#define  apply_member_fn(object,fn)    ((object).*(fn))
//...
apply_member_fn(frob,fn_ptr[i])(x, y)
```

Q113: How can I insert/access/change elements from a linked list/hashtable/etc?

A: I'll use a `inserting into a linked list' as a prototypical example. The obvious approach is to allow insertion at the head and tail of the list, but that would produce a library that is too weak (a weak library is almost worse than no library). Whenever encapsulation frustrates rather than helps a user, it may be that the class' public interface needs enhancing. If class List only supports adding at the front and tail, it *definitely* needs more strength.

This answer will be a lot to swallow for novice C++'ers, so I'll give a couple of options. As usual, the first is easiest, while the second option is better. I also give a thumbnail sketch of a third option, which has certain advantages and disadvantages over the second option.

[1] Empower the List with a `viewport' or `cursor' that references an arbitrary list element. Implies adding member fns to List such as advance(), backup(), atend(), atbegin(), rewind(), fastforward(), and current(). `current()' returns the element of the List that is currently `under the cursor'. Finally you'll need to add a few more member fns to *mutate* the list, such as changeto(X), insert(X), remove(), etc.

[2] Provide a separate class called ListIter. ListIter has member fns named similar to the above (though probably with operator overloading, but that's just syntactic sugar for member fns). The List itself would have none of the above mentioned member fns, and the ListIter would be a `friend' of List (ListIter would have to have access to the innards of List; this allows the world to have safe access abilities to List without violating encapsulation).

The reason option [2] is better becomes apparent when you use classes that only support [1]. In particular, if you pass a List off to a subcall, you'd better hope the subcall didn't warp the `cursor' around, or your code may fail. Also, if you try to get all pairs of elements, it's very hard if you only have one cursor. The distinct-class iterator concept removes these restrictions. My own class library uses these extensively, as will most any other commercial grade class library.

Note that the options are not mutually exclusive; it is possible to provide both [2] *and* [1], giving rise to the notion of a `primary' list position, with instances of ListIter being somewhat secondary.

[3] The third possibility considers the entire iteration as an atomic event. A class is created which embodies this event. The nice thing about this third alternative is that the public access methods (which may be virtual fns) can be avoided during the inner loop, thus enhancing performance. The down side is that you get extra object code in the application, since templates gain speed by duplicating code. This third technique is due to Andrew Koenig in a paper published recently in JOOP [`Templates as interfaces', JOOP, 4, 5 (Sept 91)]. You can also see a taste of it in Bjarne Stroustrup's book, The C++ Programming Language Second Edition (look for `Comparator' in the index).

Q114: What's the idea behind `templates'?

A: A template is a cookie-cutter that specifies how to cut cookies that all look pretty much the same (although the cookies can be made of various kinds of dough, they'll all have the same basic shape). In the same way, a class template is a cookie cutter to description of how to build classes that all look basically the same, and a function template describes how to build similar looking functions.

The questions about templates are in the `Containers' section since templates are often used to build type safe containers (although this only scratches the surface for how they can be used). We will see how to use templates to build container classes below.

Q115: What's the syntax / semantics for a `function template'?

A: Consider this function that swaps its two integer arguments:

```
void swap(int& x, int& y)
{
  int tmp = x;
  x = y;
  y = tmp;
}
```

If we also had to swap floats, longs, Strings, Sets, and FileSystems, we'd get pretty tired of coding lines that look almost identical except for the type. Mindless repetition is an ideal job for a computer, hence a function template:

```
template<class T>
void swap(T& x, T& y)
{
  T tmp = x;
  x = y;
  y = tmp;
}
```

Every time we used `swap()' with a given pair of types, the compiler will go to the above definition and will create yet another `template function' as an instantiation of the above. Ex:

```
main()
{
  int    i,j;  /*...*/  swap(i,j);  //instantiates a swap for `int'
  float  a,b;  /*...*/  swap(a,b);  //instantiates a swap for `float'
  char   c,d;  /*...*/  swap(c,d);  //instantiates a swap for `char'
  String s,t;  /*...*/  swap(s,t);  //instantiates a swap for `String'
}
```

(note: a `template function' is the instantiation of a `function template').

Q116: What's the syntax / semantics for a `class template'?

A: Consider this container class of that acts like an array of integers:

```
//this would go into a header file such as `Vec.h':
class Vec {
public:
  int         len()             const { return xlen;     }
  const int&  operator[](int i) const { xdata[check(i)]; }
        int&  operator[](int i)       { xdata[check(i)]; }
              Vec(int L=10) : xlen(L), xdata(new int[L]) { /*verify*/ }
             ~Vec()                   { delete [] xdata; }
private:
  int  xlen;
  int* xdata;
  int  check(int i);  //return i if i>=0 && i<xlen else throw exception
};

//this would be part of a `.C' file such as `Vec.C':
int Vec::check(int i)
{
  if (i < 0 || i >= xlen) throw BoundsViol("Vec", i, xlen);
  return i;
}
```

Just as with `swap()' above, repeating the above over and over for Vec of float, char, String, Vec, Vec-of-Vec-of-Vec, etc, will become tedious. Hence we create a single class template:

```
//this would go into a header file such as `Vec.h':
template<class T>
class Vec {
public:
  int       len()             const { return xlen;     }
  const T&  operator[](int i) const { xdata[check(i)]; }
        T&  operator[](int i)       { xdata[check(i)]; }
            Vec(int L=10) : xlen(L), xdata(new T[L]) { /*verify*/ }
           ~Vec()                   { delete [] xdata; }
private:
  int  xlen;
  T*   xdata;
  int  check(int i);  //return i if i>=0 && i<xlen else throw exception
};

//this would be part of a `.C' file such as `Vec.C':
template<class T>
int Vec<T>::check(int i)
{
  if (i < 0 || i >= xlen) throw BoundsViol("Vec", i, xlen);
  return i;
}
```

Unlike template functions, template classes (instantiations of class templates) need to be explicit about the parameters over which they are instantiating:

```
main()
{
```

```
  Vec<int>        vi;
  Vec<float>      vf;
  Vec<char*>      vc;
  Vec<String>     vs;
  Vec< Vec<int> > vv;
}          // ^^^-- note the space; do NOT use Vec<Vec<int>> since the
           //       `maximal munch' rule would grab a single `>>' token
```

Q117: What is a `parameterized type'?

A: A parameterized type is a type that is parameterized over another value or type. Ex: List<int> is a type that is parameterized over another type, `int'. Therefore the C++ rendition of parameterized types is provided by class templates.

Q118: What is `genericity'?

A: Not to be confused with `generality' (which just means avoiding solutions which are overly specific), `genericity' means parameterized types. In C++, this is provided by class templates.

Q119: How can I fake templates if I don't have a compiler that supports them?

A: The best answer is: buy a compiler that supports templates. When this is not feasible, the next best answer is to buy or build a template preprocessor (ex: reads C++-with-templates, outputs C++-with-expanded-template-classes; such a system needn't be perfect; it cost my company about three man-weeks to develop such a preprocessor). If neither of these is feasible, you can use the macro preprocessor to fake templates. But beware: it's tedious; templates are a better solution wrt development and maintenance costs.

Here's how you'd declare my `Vec' example from above. First we define `Vec(T)' to concatenate the name `Vec' with that of the type T (ex: Vec(String) becomes `VecString'). This would go into a header file such as Vec.h:

```
#include <generic.h>    //to get the `name2()' macro
#define  Vec(T)  name2(Vec,T)
```

Next we declare the class Vec(T) using the name `Vecdeclare(T)' (in general you would postfix the name of the class with `declare', such as `Listdeclare' etc):

```
#define Vecdeclare(T)                                       \
class Vec(T) {                                              \
  int  xlen;                                                \
  T*   xdata;                                               \
  int  check(int i);  /*return i if in bounds else throw*/  \
public:                                                     \
  int       len()             const { return xlen;     }    \
  const T&  operator[](int i) const { xdata[check(i)]; }    \
        T&  operator[](int i)       { xdata[check(i)]; }    \
            Vec(T)(int L=10): xlen(L), xdata(new T[L]) {/*...*/}\
           ~Vec(T)()                   { delete [] xdata; } \
};
```

Note how each occurrence of `Vec' has the `(T)' postfixed. Finally we set up another macro that `implements' the non-inline member function(s) of Vec:

```
//strangely enough this can also go into Vec.h
#define Vecimplement(T)                                     \
int Vec(T)::check(int i)                                    \
{                                                           \
  if (i < 0 || i >= xlen) throw BoundsViol("Vec", i, xlen); \
  return i;                                                 \
}
```

When you wish to use a Vec-of-String and Vec-of-int, you would say:

```
#include "Vec.h"     //pulls in <generic.h> too; see below...
declare(Vec,String)  //`declare()' is a macro defined in <generic.h>
declare(Vec,int)
Vec(String) vs;      //Vec(String) becomes the single token `VecString'
Vec(int)    vi;
```

In exactly one source file in the system, you must provide implementations for the non-inlined member functions:

```
#include "Vec.h"
declare  (Vec,String)   declare  (Vec,int)   declare  (Vec,float)
implement(Vec,String)   implement(Vec,int)   implement(Vec,float)
```

Note that types whose names are other than a single identifier do not work properly. Ex: Vec(char*) creates a class whose name is `Vecchar*'. The patch is to create a typedef for the appropriate type:

```
#include "Vec.h"
typedef char* charP;
```

```
declare(Vec,charP)
```

It is important that every declaration of what amounts to `Vec<char*>' must all use exactly the same typedef, otherwise you will end up with several equivalent classes, and you'll have unnecessary code duplication. This is the sort of tedium which a template mechanism can handle for you.

Q120: Why don't variable arg lists work for C++ on a Sun SPARCstation?

A: That is a bug in cfront 2.1. There is a magic variable, __builtin_va_alist. It has to be added to the end of the argument list of a varadic function, and it has to be referenced in va_start. This requires source modification to (a) print `,__builtin_va_alist' on the end of the argument list, (b) pretend that this arg was declared, and thus pass references to it, and (c) not mangle its name in the process.

Here's a tiny bit more detail:

1) when print2.c prints a varadic procedure, it should add __builtin_va_alist to the end. It need not declare it. Type of int to the Sun C compiler (the default) is fine.

2) #define va_start(ap,parmN) ap = (char *)&__builtin_va_alist

3) when find.c see this reference to __builtin_va_alist, it should construct a special cfront name node. When name::print sees that node it should print its name literally, without adding any mangling.

4) the same trick is needed, with the name va_alist, for Ultrix/MIPS and HPUX.

5) the net result, in generated C code, looks like:

```
void var_proc (a, b, c, __builtin_va_alist)
  int a;
  float b;
  char * c;
{
    char * val;
    val = &__builtin_va_alist;
    ...
}
```

Q121: GNU C++ (g++) produces big executables for tiny programs; Why?

A: libg++ (the library used by g++) was probably compiled with debug info (-g). On some machines, recompiling libg++ without debugging can save lots of disk space (~1 Meg; the down-side: you'll be unable to trace into libg++ calls). Merely `strip'ping the executable doesn't reclaim as much as recompiling without -g followed by subsequent `strip'ping the resultant `a.out's.

Use `size a.out' to see how big the program code and data segments really are rather than `ls -s a.out' which includes the symbol table.

Q122: Is there a yacc-able C++ grammar?

A: Jim Roskind is the author of a yacc grammar for C++. It's roughly compatible with the portion of the language implemented by USL cfront 2.0. Jim's grammar deviates from cfront (and the ARM) in a couple of what I understand to be minor ways. Ultimately any deviation from a standard is unthinkable, but the number of real programs that are interpreted differently is relatively small, so the `consequence' of the deviation is not great.

I have used Jim's grammar when a grad student wrote a C++ templatizer mechanism (reads ANSI-C++, spits out pre-templates C++), but my expertise does not include precise knowledge of where the grammar deviates from `official'. I have found it to parse most things correctly, but I am aware that there are differences. (is that noncommittal enough? :-)

The grammar can be accessed by anonymous ftp from the following sites:

* ics.uci.edu (128.195.1.1) in the ftp/gnu directory (even though
  neither of the archives are GNU related) as:
      c++grammar2.0.tar.Z and byacc1.8.tar.Z
* mach1.npac.syr.edu (128.230.7.14) in the ftp/pub/C++ directory as:
      c++grammar2.0.tar.Z and byacc1.8.tar.z

Jim Roskind can be reached at jar@hq.ileaf.com or ...!uunet!leafusa!jar

Q123: What is C++ 1.2? 2.0? 2.1? 3.0?

A: These are not versions of the language, but rather versions of the USL translator, cfront. However many people discuss these as levels of the language as well, since each of the above versions represents additions to the portion of the C++ language which is implemented by the compiler. When ANSI/ISO C++ is finalized, conformance with the ANSI/ISO spec will become more important than conformance with cfront version X.Y, but presently, cfront is acting as a de facto standard to help coordinate the industry (although it leaves certain features of the language unimplemented as well).

VERY roughly speaking, these are the major features:
 * 2.0 includes multiple/virtual inheritance and pure virtual functions.
 * 2.1 includes semi-nested classes and `delete [] ptr_to_array'.
 * 3.0 includes fully-nested classes, templates and `i++' vs `++i'.
 *?4.0? will include exceptions.

Q124: How does the lang accepted by cfront 3.0 differ from that accepted by 2.1?

A: USL cfront release 3.0 provides implementations of a number of features that were previously flagged as `sorry not implemented':

templates, fully nested types, prefix and postfix increment and decrement operators, initialization of single dimension arrays of class objects with ctors taking all default arguments, use of operators &&, ||, and ?: with expressions requiring temporaries of class objects containing destructors, and implicit named return values.

The major feature not accepted by cfront 3.0 is exceptions.

Q125: Why are exceptions going to be implemented after templates? Why not both?

A: Most C++ compiler vendors are providing templates in their present release, but very few are providing exceptions (I know of only one). The reason for going slowly is that both templates and exception handling are difficult to implement *well*. A poor template implementation will give slow compile and link times and a poor exception handling implementation will give slow run times. Good implementations will not have those problems.

C++ compiler vendors are human beings too, and they can only get so much done at once. However they know that the C++ community is craving for the `whole' language, so they'll be pushing hard to fill that need.

Q126: What was C++ 1.xx, and how is it different from the current C++ language?

A: C++ 1.2 (the version number corresponds to the release number of USL's cfront) corresponds to Bjarne Stroustrup's first edition (`The C++ Programming Language', ISBN 0-201-12078-X). In contrast, the present version (3.0) corresponds roughly to the `ARM', except for exceptions. Here is a summary of the differences between the first edition of Bjarne's book and the ARM:

- A class can have more than one direct base class (multiple inheritance).
- Class members can be `protected'.
- Pointers to class members can be declared and used.
- Operators `new' and `delete' can be overloaded and declared for a class. This allows the "assignment to `this'" technique for class specific specific storage management to be removed to the anachronism section
- Objects can be explicitly destroyed.
- Assignment and Copy-Initialization default to member-wise assignment and copy-initialization.
- The `overload' keyword was made redundant and removed to the anachronism section.
- General expressions are allowed as initializers for static objects.
- Data objects can be `volatile' (new keyword).
- Initializers are allowed for `static' class members.
- Member functions can be `static'.
- Member functions can be `const' or `volatile'.
- Linkage to non-C++ program fragments can be explicitly declared.
- Operators `->', `->*' and `,' can be overloaded.
- Classes can be abstract.
- Prefix and postfix application of `++' and `--' on a user-defined type can be distinguished.
- Templates.
- Exception handling.

Q127: Why are classes with static data members getting linker errors?

A: Static member variables must be given an explicit definition in exactly one module. Ex:

```
class X {
  //...
  static int i;  //*declare* static member X::i
  //...
};
```

The linker will holler at you (`X::i is not defined') unless (exactly) one of your source files has something like the following:

```
int X::i = some_expression_evaluating_to_an_int;     //*define* X::i
```

or:

```
int X::i;       //define --but don't initialize-- X::i
```

The usual place to define static member variables of class `X' is file `X.C' (or X.cpp, X.cc, X.c++, X.c or X.cxx; see question on file naming conventions).

Q128: What's the difference between the keywords struct and class?

A: The members and base classes of a struct are public by default, while in class, they default to private. Base classes of a struct are public by default while they are private by default with `class' (however you should make your base classes *explicitly* public, private, or protected). `Struct' and `class' are otherwise functionally equivalent.

Q129: Why can't I overload a function by its return type?

Ex: the compiler says the following two are an error:

```
char  f(int i);
float f(int i);
```

A: Return types are not considered when determining unique signatures for overloading functions; only the number and type of parameters are considered. Reason: which function should be called if the return value is ignored? Ex:

```
main()
{
  f(3);  //which should be invoked??
}
```

Q130: What is `persistence'?  What is a `persistent object'?

A: Loosely speaking, a persistent object is one that lives on after the program which created it has stopped. Persistent objects can even outlive various versions of the creating program, can outlive the disk system, the operating system, or even the hardware on which the OS was running when they were created.

The challenge with persistent objects is to effectively store their method code out on secondary storage along with their data bits (and the data bits and method code of all subobjects, and of all their subobjects, etc). This is non-trivial when you have to do it yourself. In C++, you have to do it yourself. C++/OO databases can help hide the mechanism for all this.

Q131: Is there a TeX or LaTeX macro that fixes the spacing on `C++'?

A: Yes, here are two:

```
\def\CC{C\raise.22ex\hbox{{\footnotesize +}}\raise.22ex\hbox{\footnotesize +}}
```

```
\def\CC{{C\hspace{-.05em}\raisebox{.4ex}{\tiny\bf ++}}}
```

Q132: Where can I access C++2LaTeX, a LaTeX pretty printer for C++ source?

A: Here are a few ftp locations:

Host aix370.rrz.uni-koeln.de (134.95.80.1) Last updated 15:41 26 Apr 1991
Location: /tex
FILE rw-rw-r-- 59855 May 5 1990 C++2LaTeX-1.1.tar.Z

Host utsun.s.u-tokyo.ac.jp (133.11.11.11) Last updated 05:06 20 Apr 1991
Location: /TeX/macros
FILE rw-r--r-- 59855 Mar 4 08:16 C++2LaTeX-1.1.tar.Z

Host nuri.inria.fr (128.93.1.26) Last updated 05:23 9 Apr 1991
Location: /TeX/tools
FILE rw-rw-r-- 59855 Oct 23 16:05 C++2LaTeX-1.1.tar.Z

Host iamsun.unibe.ch (130.92.64.10) Last updated 05:06 4 Apr 1991
Location: /TeX
FILE rw-r--r-- 59855 Apr 25 1990 C++2LaTeX-1.1.tar.Z

Host iamsun.unibe.ch (130.92.64.10) Last updated 05:06 4 Apr 1991
Location: /TeX
FILE rw-r--r-- 51737 Apr 30 1990
C++2LaTeX-1.1-PL1.tar.Z

Host tupac-amaru.informatik.rwth-aachen.de (192.35.229.9) Last updated 05:07 18 Apr 1991
Location: /pub/textproc/TeX
FILE rw-r--r-- 72957 Oct 25 13:51 C++2LaTeX-1.1-PL4.tar.Z

Host wuarchive.wustl.edu (128.252.135.4) Last updated 23:25 30 Apr 1991
Location: /packages/tex/tex/192.35.229.9/textproc/TeX
FILE rw-rw-r-- 49104 Apr 10 1990 C++2LaTeX-PL2.tar.Z
FILE rw-rw-r-- 25835 Apr 10 1990 C++2LaTeX.tar.Z

Host tupac-amaru.informatik.rwth-aachen.de (192.35.229.9) Last updated 05:07 18 Apr 1991
Location: /pub/textproc/TeX
FILE rw-r--r-- 74015 Mar 22 16:23 C++2LaTeX-1.1-PL5.tar.Z
Location: /pub
FILE rw-r--r-- 74015 Mar 22 16:23 C++2LaTeX-1.1-PL5.tar.Z

Host sol.cs.ruu.nl (131.211.80.5) Last updated 05:10 15 Apr 1991
Location: /TEX/TOOLS
FILE rw-r--r-- 74015 Apr 4 21:02x C++2LaTeX-1.1-PL5.tar.Z

Host tupac-amaru.informatik.rwth-aachen.de (192.35.229.9) Last updated 05:07 18 Apr 1991
Location: /pub/textproc/TeX
FILE rw-r--r-- 4792 Sep 11 1990 C++2LaTeX-1.1-patch#1
FILE rw-r--r-- 2385 Sep 11 1990 C++2LaTeX-1.1-patch#2
FILE rw-r--r-- 5069 Sep 11 1990 C++2LaTeX-1.1-patch#3
FILE rw-r--r-- 1587 Oct 25 13:58 C++2LaTeX-1.1-patch#4
FILE rw-r--r-- 8869 Mar 22 16:23 C++2LaTeX-1.1-patch#5
FILE rw-r--r-- 1869 Mar 22 16:23 C++2LaTeX.README

Host rusmv1.rus.uni-stuttgart.de (129.69.1.12) Last updated 05:13 13 Apr 1991
Location: /soft/tex/utilities
FILE rw-rw-r-- 163840 Jul 16 1990 C++2LaTeX-1.1.tar

Q133: Where can I access `tgrind', a pretty printer for C++/C/etc source?

A: `tgrind' reads the file to be printed and a command line switch to see what the source language is. It then reads a language definition database file and learns the syntax of the language (list of keywords, literal string delimiters, comment delimiters, etc).

`tgrind' usually comes with the public distribution of TeX and LaTeX. Look in the directory:

...tex82/contrib/van/tgrind

A more up-to-date version of tgrind by Jerry Leichter can be found on:

venus.ycc.yale.edu in [.TGRIND]

Q134: Is there a C++-mode for GNU emacs? If so, where can I get it?

A: Yes, there is a C++-mode for GNU emacs. You can get it via:

c++-mode-2 (1.0) 87-12-08

Bruce Eckel,Thomas Keffer, <eckel@beluga.ocean.washington.edu,uw-beaver!beluga!eckel,keffer@sperm.ocean.washington.edu>
archive.cis.ohio-state.edu:/pub/gnu/emacs/elisp-archive/as-is/c++-mode-2.el.Z

Another C++ major mode.

c++-mode 89-11-07

Dave Detlefs, et al, <dld@cs.cmu.edu>
archive.cis.ohio-state.edu:/pub/gnu/emacs/elisp-archive/modes/c++-mode.el.Z
C++ major mode.

c++ 90-02-01

David Detlefs, Stewart Clamen, <dld@f.gp.cs.cmu.edu,clamen@cs.cmu.edu>
archive.cis.ohio-state.edu:/pub/gnu/emacs/elisp-archive/modes/c++.el.Z
C++ code editing commands for Emacs

c-support (46) 89-11-04

Lynn Slater, <lrs@indetech.com>
archive.cis.ohio-state.edu:/pub/gnu/emacs/elisp-archive/misc/c-support.el.Z
Partial support for team C/C++ development.

Q135: What is `InterViews'?

A: A non-proprietary toolkit for graphic user interface programming in C++, developed by Mark Linton and others when he was at Stanford. Unlike C++ wrappers for C libraries such as Motif, InterViews is a true object library. Commercially maintained versions are available from Quest, while freely redistributable versions (running on top of X) from interviews.stanford.edu (an ftp site). Copies of this are (were?) distributed with the regular X distribution. Other sources of information include comp.windows.interviews, which has its own FAQ.

Q136: Where can I get OS-specific questions answered (ex:BC++,DOS,Windows,etc)?

A: see comp.os.msdos.programmer, BC++ and Zortech mailing lists, BC++ and Zortech bug lists, comp.windows.ms.programmer, comp.unix.programmer, etc.

You can subscribe to the BC++ mailing list by sending email to:

| To: listserv@ucf1vm.cc.ucf.edu <---or LISTSERV@UCF1VM.BITNET
| Subject: SUB TCPLUS-L
| Reply-to: you@your.email.addr <---ie: put your return address here

The BC++ bug report is available via anonymous ftp from sun.soe.clarkson.edu [128.153.12.3] from the file ~ftp/pub/Turbo-C++/bug-report (also, I post it on comp.lang.c++ on the first each month).

Relevant email addresses:

ztc-list@zortech.com	General requests and discussion
ztc-list-request@zortech.com	Requests to be added to ztc-list
ztc-bugs@zortech.com	For _short_ bug reports

Q137: Why does my DOS C++ program says `Sorry: floating point code not linked'?

A: The compiler attempts to save space in the executable by not including the float-to-string format conversion routines unless they are necessary, and sometimes does not recognize some code that does require it. Taking the address of a float in an argument list of a function call seems to trigger it, but taking the address of a float element of a struct may fool the compiler. A "%f" in a printf/scanf format string doesn't trigger it because format strings aren't examined at compile-time.

You can fix it by (1) using <iostream.h> instead of <stdio.h>, or (2) by including the following function definition somewhere in your compilation (but don't call it!):

```
static void dummyfloat(float *x) { float y; dummyfloat(&y); }
```

See question on stream I/O for more reasons to use <iostream.h> vs <stdio.h>.