Morgan Stanley | Columbia University | Churchill College, Cambridge

Bjarne Stroustrup's C++ Style and Technique FAQ

Modified September 30, 2017

These are questions about C++ Style and Technique that people ask me often. If you have better questions or comments on the answers, feel free to email me (bs at cs dot tamu dot edu). Please remember that I can't spend all of my time improving my homepages.

I have contributed to the new, unified, isocpp.org C++ FAQ maintained by The C++ Foundation of which I am a director. The maintenance of this FAQ is likely to become increasingly sporatic.

For more general questions, see my general FAQ.

For terminology and concepts, see my C++ glossary.

Please note that these are just a collection of questions and answers. They are not a substitute for a carefully selected sequence of examples and explanations as you would find in a good textbook. Nor do they offer detailed and precise specifications as you would find in a reference manual or the standard. See The Design and Evolution of C++ for questions related to the design of C++. See The C++ Programming Language for questions about the use of C++ and its standard library.

Translations:

- Chinese of some of this Q&A with annotations
- another Chinese version
- Hungarian
- Japanese
- Ukrainian

How do I write this very simple program?

Often, especially at the start of semesters, I get a lot of questions about how to write very simple programs. Typically, the problem to be solved is to read in a few numbers, do something with them, and write out an answer. Here is a sample program that does that:

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

int main()
{
        vector<double> v;

        double d;
        while(cin>>d) v.push_back(d);   // read elements
        if (!cin.eof()) {               // check if input failed
                cerr << "format error\n";
                return 1;       // error return
        }

        cout << "read " << v.size() << " elements\n";

        reverse(v.begin(),v.end());
        cout << "elements in reverse order:\n";
        for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';

        return 0; // success return
}
```

Here are a few observations about this program:

- This is a Standard ISO C++ program using the standard library. Standard library facilities are declared in namespace std in headers without a .h suffix.
- If you want to compile this on a Windows machine, you need to compile it as a "console application". Remember to give your source file the .cpp suffix or the compiler might think that it is C (not C++) source.
- Yes, main() returns an int.
- Reading into a standard vector guarantees that you don't overflow some arbitrary buffer. Reading into an array without making a "silly error" is beyond the ability of complete novices - by the time you get that right, you are no longer a complete novice. If you doubt this claim, I suggest you read my paper "Learning Standard C++ as a New Language", which you can download from my publications list.
- The !cin.eof() is a test of the stream's format. Specifically, it tests whether the loop ended by finding end-of-file (if not, you didn't get input of the expected type/format). For more information, look up "stream state" in your C++ textbook.
- A vector knows its size, so I don't have to count elements.
- Yes, I know that I could declare *i* to be a *vector<double>::size_type* rather than plain *int* to quiet warnings from some hyper-suspicious compilers, but in this case,I consider that too pedantic and distracting.
- This program contains no explicit memory management, and it does not leak memory. A vector keeps track of the memory it uses to store its elements. When a vector needs more memory for elements, it allocates more; when a vector goes out of scope, it frees that memory. Therefore, the user need not be concerned with the allocation and deallocation of memory for vector elements.
- for reading in strings, see How do I read a string from input?.
- The program ends reading input when it sees "end of file". If you run the program from the keybord on a Unix machine "end of file" is Ctrl-D. If you are on a Windows machine that because of a bug doesn't recognize an end-of-file character, you might prefer this slightly more complicated version of the program that terminates input with the word "end":

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<string>
```

```
using namespace std;

int main()
{
        vector<double> v;

        double d;
        while(cin>>d) v.push_back(d);   // read elements
        if (!cin.eof()) {               // check if input failed
                cin.clear();            // clear error state
                string s;
                cin >> s;               // look for terminator string
                if (s != "end") {
                        cerr << "format error\n";
                        return 1;       // error return
                }
        }

        cout << "read " << v.size() << " elements\n";

        reverse(v.begin(),v.end());
        cout << "elements in reverse order:\n";
        for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';

        return 0; // success return
}
```

For more examples of how to use the standard library to do simple things simply, see the "Tour of the Standard Library" Chapter of TC++PL4.

Can you recommend a coding standard?

Yes: The C++ Core Guidelines. This is an ambitious project to guide people to an effective style of modern C++ and to provide tool to support its rules. It encourages people to use C++ as a completely type- and resource-safe language without compromising performance or adding verbosity. There are videos describing the guidelines project.

The main point of a C++ coding standard is to provide a set of rules for using C++ for a particular purpose in a particular environment. It follows that there cannot be one coding standard for all uses and all users. For a given application (or company, application area, etc.), a good coding standard is better than no coding standard. On the other hand, I have seen many examples that demonstrate that a bad coding standard is worse than no coding standard.

Don't use C coding standards (even if slightly modified for C++) and don't use ten-year-old C++ coding standards (even if good for their time). C++ isn't (just) C and Standard C++ is not (just) pre-standard C++.

Why do my compiles take so long?

You may have a problem with your compiler. It may be old, you may have it installed wrongly, or your computer might be an antique. I can't help you with such problems.

However, it is more likely that the program that you are trying to compile is poorly designed, so that compiling it involves the compiler examining hundreds of header files and tens of thousands of lines of code. In principle, this can be avoided. If this problem is in your library vendor's design, there isn't much you can do (except changing to a better library/vendor), but you can structure your own code to minimize re-compilation after changes. Designs that do that are typically better, more maintainable, designs because they exhibit better separation of concerns.

Consider a classical example of an object-oriented program:

```
class Shape {
public:         // interface to users of Shapes
        virtual void draw() const;
```

```
        virtual void rotate(int degrees);
        // ...
protected:      // common data (for implementers of Shapes)
        Point center;
        Color col;
        // ...
};

class Circle : public Shape {
public:
        void draw() const;
        void rotate(int) { }
        // ...
protected:
        int radius;
        // ...
};

class Triangle : public Shape {
public:
        void draw() const;
        void rotate(int);
        // ...
protected:
        Point a, b, c;
        // ...
};
```

The idea is that users manipulate shapes through Shape's public interface, and that implementers of derived classes (such as Circle and Triangle) share aspects of the implementation represented by the protected members.

There are three serious problems with this apparently simple idea:

- It is not easy to define shared aspects of the implementation that are helpful to all derived classes. For that reason, the set of protected members is likely to need changes far more often than the public interface. For example, even though "center" is arguably a valid concept for all Shapes, it is a nuisance to have to maintain a point "center" for a Triangle - for triangles, it makes more sense to calculate the center if and only if someone expresses interest in it.
- The protected members are likely to depend on "implementation" details that the users of Shapes would rather not have to depend on. For example, many (most?) code using a Shape will be logically independent of the definition of "Color", yet the presence of Color in the definition of Shape will probably require compilation of header files defining the operating system's notion of color.
- When something in the protected part changes, users of Shape have to recompile - even though only implementers of derived classes have access to the protected members.

Thus, the presence of "information helpful to implementers" in the base class that also acts as the interface to users is the source of instability in the implementation, spurious recompilation of user code (when implementation information changes), and excess inclusion of header files into user code (because the "information helpful to implementers" needs those headers). This is sometimes known as the "brittle base class problem."

The obvious solution is to omit the "information helpful to implemeters" for classes that are used as interfaces to users. That is, to make interfaces, pure interfaces. That is, to represent interfaces as abstract classes:

```
class Shape {
public:         // interface to users of Shapes
        virtual void draw() const = 0;
        virtual void rotate(int degrees) = 0;
        virtual Point center() const = 0;
        // ...

        // no data
};
```

```
class Circle : public Shape {
public:
        void draw() const;
        void rotate(int) { }
        Point center() const { return cent; }
        // ...
protected:
        Point cent;
        Color col;
        int radius;
        // ...
};

class Triangle : public Shape {
public:
        void draw() const;
        void rotate(int);
        Point center() const;
        // ...
protected:
        Color col;
        Point a, b, c;
        // ...
};
```

The users are now insulated from changes to implementations of derived classes. I have seen this technique decrease build times by orders of magnitudes.

But what if there really is some information that is common to all derived classes (or simply to several derived classes)? Simply make that information a class and derive the implementation classes from that also:

```
class Shape {
public:         // interface to users of Shapes
        virtual void draw() const = 0;
        virtual void rotate(int degrees) = 0;
        virtual Point center() const = 0;
        // ...

        // no data
};

struct Common {
        Color col;
        // ...
};

class Circle : public Shape, protected Common {
public:
        void draw() const;
        void rotate(int) { }
        Point center() const { return cent; }
        // ...
protected:
        Point cent;
        int radius;
};

class Triangle : public Shape, protected Common {
public:
        void draw() const;
        void rotate(int);
        Point center() const;
        // ...
protected:
        Point a, b, c;
};
```

Why is the size of an empty class not zero?

To ensure that the addresses of two different objects will be different. For the same reason, "new"

always returns pointers to distinct objects. Consider:

```
class Empty { };

void f()
{
        Empty a, b;
        if (&a == &b) cout << "impossible: report error to compiler supplier";

        Empty* p1 = new Empty;
        Empty* p2 = new Empty;
        if (p1 == p2) cout << "impossible: report error to compiler supplier";
}
```

There is an interesting rule that says that an empty base class need not be represented by a separate byte:

```
struct X : Empty {
        int a;
        // ...
};

void f(X* p)
{
        void* p1 = p;
        void* p2 = &p->a;
        if (p1 == p2) cout << "nice: good optimizer";
}
```

This optimization is safe and can be most useful. It allows a programmer to use empty classes to represent very simple concepts without overhead. Some current compilers provide this "empty base class optimization".

Why do I have to put the data in my class declarations?

You don't. If you don't want data in an interface, don't put it in the class that defines the interface. Put it in derived classes instead. See, Why do my compiles take so long?.

Sometimes, you do want to have representation data in a class. Consider class complex:

```
template<class Scalar> class complex {
public:
        complex() : re(0), im(0) { }
        complex(Scalar r) : re(r), im(0) { }
        complex(Scalar r, Scalar i) : re(r), im(i) { }
        // ...

        complex& operator+=(const complex& a)
                { re+=a.re; im+=a.im; return *this; }
        // ...
private:
        Scalar re, im;
};
```

This type is designed to be used much as a built-in type and the representation is needed in the declaration to make it possible to create genuinely local objects (i.e. objects that are allocated on the stack and not on a heap) and to ensure proper inlining of simple operations. Genuinely local objects and inlining is necessary to get the performance of complex close to what is provided in languages with a built-in complex type.

Why are member functions not virtual by default?

Because many classes are not designed to be used as base classes. For example, see class complex.

Also, objects of a class with a virtual function require space needed by the virtual function call mechanism - typically one word per object. This overhead can be significant, and can get in the

way of layout compatibility with data from other languages (e.g. C and Fortran).

See The Design and Evolution of C++ for more design rationale.

Why are destructors not virtual by default?

Because many classes are not designed to be used as base classes. Virtual functions make sense only in classes meant to act as interfaces to objects of derived classes (typically allocated on a heap and accessed through pointers or references).

So when should I declare a destructor virtual? Whenever the class has at least one virtual function. Having virtual functions indicate that a class is meant to act as an interface to derived classes, and when it is, an object of a derived class may be destroyed through a pointer to the base. For example:

```
class Base {
        // ...
        virtual ~Base();
};

class Derived : public Base {
        // ...
        ~Derived();
};

void f()
{
        Base* p = new Derived;
        delete p;       // virtual destructor used to ensure that ~Derived is called
}
```

Had Base's destructor not been virtual, Derived's destructor would not have been called - with likely bad effects, such as resources owned by Derived not being freed.

Why don't we have virtual constructors?

A virtual call is a mechanism to get work done given partial information. In particular, "virtual" allows us to call a function knowing only an interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a "call to a constructor" cannot be virtual.

Techniques for using an indirection when you ask to create an object are often referred to as "Virtual constructors". For example, see TC++PL3 15.6.2.

For example, here is a technique for generating an object of an appropriate type using an abstract class:

```
struct F {      // interface to object creation functions
        virtual A* make_an_A() const = 0;
        virtual B* make_a_B() const = 0;
};

void user(const F& fac)
{
        A* p = fac.make_an_A(); // make an A of the appropriate type
        B* q = fac.make_a_B();  // make a B of the appropriate type
        // ...
}

struct FX : F {
        A* make_an_A() const { return new AX(); } // AX is derived from A
        B* make_a_B() const { return new BX();  } // BX is derived from B
};

struct FY : F {
```

```
        A* make_an_A() const { return new AY(); } // AY is derived from A
        B* make_a_B() const { return new BY();  } // BY is derived from B
};

int main()
{
        FX x;
        FY y;
        user(x);        // this user makes AXs and BXs
        user(y);        // this user makes AYs and BYs

        user(FX());     // this user makes AXs and BXs
        user(FY());     // this user makes AYs and BYs
        // ...
}
```

This is a variant of what is often called "the factory pattern". The point is that user() is completely isolated from knowledge of classes such as AX and AY.

What is a pure virtual function?

A pure virtual function is a function that must be overridden in a derived class and need not be defined. A virtual function is declared to be "pure" using the curious "=0" syntax. For example:

```
class Base {
public:
        void f1();              // not virtual
        virtual void f2();      // virtual, not pure
        virtual void f3() = 0;  // pure virtual
};

Base b; // error: pure virtual f3 not overridden
```

Here, Base is an abstract class (because it has a pure virtual function), so no objects of class Base can be directly created: Base is (explicitly) meant to be a base class. For example:

```
class Derived : public Base {
        // no f1: fine
        // no f2: fine, we inherit Base::f2
        void f3();
};

Derived d;      // ok: Derived::f3 overrides Base::f3
```

Abstract classes are immensely useful for defining interfaces. In fact, a class with only pure virtual functions is often called an interface.

You can define a pure virtual function:

```
Base::f3() { /* ... */ }
```

This is very occasionally useful (to provide some simple common implementation detail for derived classes), but Base::f3() must still be overridden in some derived class.

If you don't override a pure virtual function in a derived class, that derived class becomes abstract:

```
class D2 : public Base {
        // no f1: fine
        // no f2: fine, we inherit Base::f2
        // no f3: fine, but D2 is therefore still abstract
};

D2 d;   // error: pure virtual Base::f3 not overridden
```

Why doesn't overloading work for derived classes?

That question (in many variations) are usually prompted by an example like this:

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

class B {
public:
        int f(int i) { cout << "f(int): "; return i+1; }
        // ...
};

class D : public B {
public:
        double f(double d) { cout << "f(double): "; return d+1.3; }
        // ...
};

int main()
{
        D* pd = new D;

        cout << pd->f(2) << '\n';
        cout << pd->f(2.3) << '\n';
}
```

which will produce:

```
f(double): 3.3
f(double): 3.6
```

rather than the

```
f(int): 3
f(double): 3.6
```

that some people (wrongly) guessed.

In other words, there is no overload resolution between D and B. The compiler looks into the scope of D, finds the single function "double f(double)" and calls it. It never bothers with the (enclosing) scope of B. In C++, there is no overloading across scopes - derived class scopes are not an exception to this general rule. (See D&E or TC++PL3 for details).

But what if I want to create an overload set of all my f() functions from my base and derived class? That's easily done using a using-declaration:

```
class D : public B {
public:
        using B::f;     // make every f from B available
        double f(double d) { cout << "f(double): "; return d+1.3; }
        // ...
};
```

Give that modification, the output will be

```
f(int): 3
f(double): 3.6
```

That is, overload resolution was applied to B's f() and D's f() to select the most appropriate f() to call.

Can I use "new" just as in Java?

Sort of, but don't do it blindly and there are often superior alternatives. Consider:

```
void compute(cmplx z, double d)
{
        cmplx z2 = z+d; // c++ style
        z2 = f(z2);             // use z2

        cmplx& z3 = *new cmplx(z+d);    // Java style (assuming Java could overload +)
        z3 = f(z3);
        delete &z3;
}
```

The clumbsy use of "new" for z3 is unnecessary and slow compared with the idiomatic use of a local variable (z2). You don't need to use "new" to create an object if you also "delete" that object in the same scope; such an object should be a local variable.

Can I call a virtual function from a constructor?

Yes, but be careful. It may not do what you expect. In a constructor, the virtual call mechanism is disabled because overriding from derived classes hasn't yet happened. Objects are constructed from the base up, "base before derived".

Consider:

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

class B {
public:
        B(const string& ss) { cout << "B constructor\n"; f(ss); }
        virtual void f(const string&) { cout << "B::f\n";}
};

class D : public B {
public:
        D(const string & ss) :B(ss) { cout << "D constructor\n";}
        void f(const string& ss) { cout << "D::f\n"; s = ss; }
private:
        string s;
};

int main()
{
        D d("Hello");
}
```

the program compiles and produce

```
B constructor
B::f
D constructor
```

Note *not* D::f. Consider what would happen if the rule were different so that D::f() was called from B::B(): Because the constructor D::D() hadn't yet been run, D::f() would try to assign its argument to an uninitialized string s. The result would most likely be an immediate crash.

Destruction is done "derived class before base class", so virtual functions behave as in constructors: Only the local definitions are used - and no calls are made to overriding functions to avoid touching the (now destroyed) derived class part of the object.

For more details see D&E 13.2.4.2 or TC++PL3 15.4.3.

It has been suggested that this rule is an implementation artifact. It is not so. In fact, it would be noticeably easier to implement the unsafe rule of calling virtual functions from constructors exactly as from other functions. However, that would imply that no virtual function could be written to rely on invariants established by base classes. That would be a terrible mess.

Is there a "placement delete"?

No, but if you need one you can write your own.

Consider placement new used to place objects in a set of arenas

```
class Arena {
public:
        void* allocate(size_t);
```

```
        void deallocate(void*);
        // ...
};

void* operator new(size_t sz, Arena& a)
{
        return a.allocate(sz);
}

Arena a1(some arguments);
Arena a2(some arguments);
```

Given that, we can write

```
X* p1 = new(a1) X;
Y* p2 = new(a1) Y;
Z* p3 = new(a2) Z;
// ...
```

But how can we later delete those objects correctly? The reason that there is no built-in "placement delete" to match placement new is that there is no general way of assuring that it would be used correctly. Nothing in the C++ type system allows us to deduce that p1 points to an object allocated in Arena a1. A pointer to any X allocated anywhere can be assigned to p1.

However, sometimes the programmer does know, and there is a way:

```
template<class T> void destroy(T* p, Arena& a)
{
        if (p) {
                p->~T();                // explicit destructor call
                a.deallocate(p);
        }
}
```

Now, we can write:

```
destroy(p1,a1);
destroy(p2,a2);
destroy(p3,a3);
```

If an Arena keeps track of what objects it holds, you can even write destroy() to defend itself against mistakes.

It is also possible to define a matching operator new() and operator delete() pairs for a class hierarchy TC++PL(SE) 15.6. See also D&E 10.4 and TC++PL(SE) 19.4.5.

Can I stop people deriving from my class?

Yes, but why do you want to? There are two common answers:

- for efficiency: to avoid my function calls being virtual
- for safety: to ensure that my class is not used as a base class (for example, to be sure that I can copy objects without fear of slicing)

In my experience, the efficiency reason is usually misplaced fear. In C++, virtual function calls are so fast that their real-world use for a class designed with virtual functions does not to produce measurable run-time overheads compared to alternative solutions using ordinary function calls. Note that the virtual function call mechanism is typically used only when calling through a pointer or a reference. When calling a function directly for a named object, the virtual function class overhead is easily optimized away.

If there is a genuine need for "capping" a class hierarchy to avoid virtual function calls, one might ask why those functions are virtual in the first place. I have seen examples where performance-critical functions had been made virtual for no good reason, just because "that's the way we usually do it".

The other variant of this problem, how to prevent derivation for logical reasons, has a solution in

C++11. For example:

```
struct Base {
        virtual void f();
};

struct Derived final : Base {   // now Derived is final; you cannot derive from it
        void f() override;
};

struct DD: Derived {// error: Derived is final

        // ...
};
```

For older compilers, you can use a somewhat clumsy technique:

```
class Usable;

class Usable_lock {
        friend class Usable;
private:
        Usable_lock() {}
        Usable_lock(const Usable_lock&) {}
};

class Usable : public virtual Usable_lock {
        // ...
public:
        Usable();
        Usable(char*);
        // ...
};

Usable a;

class DD : public Usable { };

DD dd;  // error: DD::DD() cannot access
        // Usable_lock::Usable_lock(): private  member
```

Why doesn't C++ provide heterogenous containers?

The C++ standard library provides a set of useful, statically type-safe, and efficient containers. Examples are vector, list, and map:

```
vector<int> vi(10);
vector<Shape*> vs;
list<string> lst;
list<double> l2
map<string,Record*> tbl;
map< Key,vector<Record*> > t2;
```

These containers are described in all good C++ textbooks, and should be preferred over arrays and "home cooked" containers unless there is a good reason not to.

These containers are homogeneous; that is, they hold elements of the same type. If you want a container to hold elements of several different types, you must express that either as a union or (usually much better) as a container of pointers to a polymorphic type. The classical example is:

```
vector<Shape*> vi;      // vector of pointers to Shapes
```

Here, vi can hold elements of any type derived from Shape. That is, vi is homogeneous in that all its elements are Shapes (to be precise, pointers to Shapes) and heterogeneous in the sense that vi can hold elements of a wide variety of Shapes, such as Circles, Triangles, etc.

So, in a sense all containers (in every language) are homogenous because to use them there must be a common interface to all elements for users to rely on. Languages that provide containers deemed heterogenous simply provide containers of elements that all provide a standard interface.

For example, Java collections provide containers of (references to) Objects and you use the (common) Object interface to discover the real type of an element.

The C++ standard library provides homogeneous containers because those are the easiest to use in the vast majority of cases, gives the best compile-time error message, and imposes no unnecessary run-time overheads.

If you need a heterogeneous container in C++, define a common interface for all the elements and make a container of those. For example:

```
class Io_obj { /* ... */ };     // the interface needed to take part in object I/O

vector<Io_obj*> vio;            // if you want to manage the pointers directly
vector< Handle<Io_obj> > v2;    // if you want a "smart pointer" to handle the objects
```

Don't drop to the lowest level of implementation detail unless you have to:

```
vector<void*> memory;   // rarely needed
```

A good indication that you have "gone too low level" is that your code gets littered with casts.

Using an Any class, such as Boost::Any, can be an alternative in some programs:

```
vector<Any> v;
```

Why are the standard containers so slow?

They are not. Probably "compared to what?" is a more useful answer. When people complain about standard-library container performance, I usually find one of three genuine problems (or one of the many myths and red herrings):

- I suffer copy overhead
- I suffer slow speed for lookup tables
- My hand-coded (intrusive) lists are much faster than std::list

Before trying to optimize, consider if you have a genuine performance problem. In most of cases sent to me, the performance problem is theoretical or imaginary: First measure, then optimise only if needed.

Let's look at those problems in turn. Often, a vector<X> is slower than somebody's specialized My_container<X> because My_container<X> is implemented as a container of pointers to X. The standard containers hold copies of values, and copy a value when you put it into the container. This is essentially unbeatable for small values, but can be quite unsuitable for huge objects:

```
vector<int> vi;
vector<Image> vim;
// ...
int i = 7;
Image im("portrait.jpg");       // initialize image from file
// ...
vi.push_back(i);        // put (a copy of) i into vi
vim.push_back(im);      // put (a copy of) im into vim
```

Now, if portrait.jpg is a couple of megabytes and Image has value semantics (i.e., copy assignment and copy construction make copies) then vim.push_back(im) will indeed be expensive. But -- as the saying goes -- if it hurts so much, just don't do it. Instead, either use a container of handles or a containers of pointers. For example, if Image had reference semantics, the code above would incur only the cost of a copy constructor call, which would be trivial compared to most image manipulation operators. If some class, say Image again, does have copy semantics for good reasons, a container of pointers is often a reasonable solution:

```
vector<int> vi;
vector<Image*> vim;
// ...
Image im("portrait.jpg");       // initialize image from file
// ...
```

```
vi.push_back(7);        // put (a copy of) 7 into vi
vim.push_back(&im);     // put (a copy of) &im into vim
```

Naturally, if you use pointers, you have to think about resource management, but containers of pointers can themselves be effective and cheap resource handles (often, you need a container with a destructor for deleting the "owned" objects).

The second frequently occuring genuine performance problem is the use of a map<string,X> for a large number of (string,X) pairs. Maps are fine for relatively small containers (say a few hundred or few thousand elements -- access to an element of a map of 10000 elements costs about 9 comparisons), where less-than is cheap, and where no good hash-function can be constructed. If you have lots of strings and a good hash function, use a hash table. The unordered_map from the standard committee's Technical Report is now widely available and is far better than most people's homebrew.

Sometimes, you can speed up things by using (const char*,X) pairs rather than (string,X) pairs, but remember that < doesn't do lexicographical comparison for C-style strings. Also, if X is large, you may have the copy problem also (solve it in one of the usual ways).

Intrusive lists can be really fast. However, consider whether you need a list at all: a vector is more compact and is therefore smaller and faster in many cases - even when you do inserts and erases. For example, if you logically have a list of a few integer elements, a vector is significantly faster than a list (any list). Also, intrusive lists cannot hold built-in types directly (an int does not have a link member). So, assume that you really need a list and that you can supply a link field for every element type. The standard-library list by default performs an allocation followed by a copy for each operation inserting an element (and a deallocation for each operation removing an element). For std::list with the default allocator, this can be significant. For small elements where the copy overhead is not significant, consider using an optimized allocator. Use a hand-crafted intrusive lists only where a list and the last ounce of performance is needed.

People sometimes worry about the cost of std::vector growing incrementally. I used to worry about that and used reserve() to optimize the growth. After measuring my code and repeatedly having trouble finding the performance benefits of reserve() in real programs, I stopped using it except where it is needed to avoid iterator invalidation (a rare case in my code). Again: measure before you optimize.

Does "friend" violate encapsulation?

No. It does not. "Friend" is an explicit mechanism for granting access, just like membership. You cannot (in a standard conforming program) grant yourself access to a class without modifying its source. For example:

```
class X {
        int i;
public:
        void m();               // grant X::m() access
        friend void f(X&);      // grant f(X&) access
        // ...
};

void X::m() { i++; /* X::m() can access X::i */ }

void f(X& x) { x.i++; /* f(X&) can access X::i */ }
```

For a description on the C++ protection model, see D&E sec 2.10 and TC++PL sec 11.5, 15.3, and C.11.

Why doesn't my constructor work right?

This is a question that comes in many forms. Such as:

- Why does the compiler copy my objects when I don't want it to?
- How do I turn off copying?
- How do I stop implicit conversions?
- How did my int turn into a complex number?

By default a class is given a copy constructor and a copy assignment that copy all elements. For example:

```
struct Point {
        int x,y;
        Point(int xx = 0, int yy = 0) :x(xx), y(yy) { }
};

Point p1(1,2);
Point p2 = p1;
```

Here we get p2.x==p1.x and p2.y==p1.y. That's often exactly what you want (and essential for C compatibility), but consider:

```
class Handle {
private:
        string name;
        X* p;
public:
        Handle(string n)
                :name(n), p(0) { /* acquire X called "name" and let p point to it */ }
        ~Handle() { delete p; /* release X called "name" */ }
        // ...
};

void f(const string& hh)
{
        Handle h1(hh);
        Handle h2 = h1; // leads to disaster!
        // ...
}
```

Here, the default copy gives us h2.name==h1.name and h2.p==h1.p. This leads to disaster: when we exit f() the destructors for h1 and h2 are invoked and the object pointed to by h1.p and h2.p is deleted twice.

How do we avoid this? The simplest solution is to prevent copying by making the operations that copy private:

```
class Handle {
private:
        string name;
        X* p;

        Handle(const Handle&);  // prevent copying
        Handle& operator=(const Handle&);
public:
        Handle(string n)
                :name(n), p(0) { /* acquire the X called "name" and let p point to it */ }
        ~Handle() { delete p; /* release X called "name" */ }
        // ...
};

void f(const string& hh)
{
        Handle h1(hh);
        Handle h2 = h1; // error (reported by compiler)
        // ...
}
```

If we need to copy, we can of course define the copy initializer and the copy assignment to provide the desired semantics.

Now return to Point. For Point the default copy semantics is fine, the problem is the constructor:

```
struct Point {
        int x,y;
```

```
        Point(int xx = 0, int yy = 0) :x(xx), y(yy) { }
};

void f(Point);

void g()
{
        Point orig;     // create orig with the default value (0,0)
        Point p1(2);    // create p1 with the default y-coordinate 0
        f(2);           // calls Point(2,0);
}
```

People provide default arguments to get the convenience used for orig and p1. Then, some are surprised by the conversion of 2 to Point(2,0) in the call of f(). A constructor taking a single argument defines a conversion. By default that's an implicit conversion. To require such a conversion to be explicit, declare the constructor explicit:

```
struct Point {
        int x,y;
        explicit Point(int xx = 0, int yy = 0) :x(xx), y(yy) { }
};

void f(Point);

void g()
{
        Point orig;     // create orig with the default value (0,0)
        Point p1(2);    // create p1 with the default y-coordinate 0
                        // that's an explicit call of the constructor
        f(2);           // error (attmpted implicit conversion)
        Point p2 = 2;   // error (attmpted implicit conversion)
        Point p3 = Point(2);    // ok (explicit conversion)
}
```

Why does C++ have both pointers and references?

C++ inherited pointers from C, so I couldn't remove them without causing serious compatibility problems. References are useful for several things, but the direct reason I introduced them in C++ was to support operator overloading. For example:

```
void f1(const complex* x, const complex* y)     // without references
{
        complex z = *x+*y;      // ugly
        // ...
}

void f2(const complex& x, const complex& y)     // with references
{
        complex z = x+y;        // better
        // ...
}
```

More generally, if you want to have both the functionality of pointers and the functionality of references, you need either two different types (as in C++) or two different sets of operations on a single type. For example, with a single type you need both an operation to assign to the object referred to and an operation to assign to the reference/pointer. This can be done using separate operators (as in Simula). For example:

```
Ref<My_type> r :- new My_type;
r := 7;                 // assign to object
r :- new My_type;       // assign to reference
```

Alternatively, you could rely on type checking (overloading).For example:

```
Ref<My_type> r = new My_type;
r = 7;                  // assign to object
r = new My_type;        // assign to reference
```

Should I use call-by-value or call-by-reference?

That depends on what you are trying to achieve:

- If you want to change the object passed, call by reference or use a pointer; e.g. void f(X&); or void f(X*);
- If you don't want to change the object passed and it is big, call by const reference; e.g. void f(const X&);
- Otherwise, call by value; e.g. void f(X);

What do I mean by "big"? Anything larger than a couple of words.

Why would I want to change an argument? Well, often we have to, but often we have an alternative: produce a new value. Consider:

```
void incr1(int& x);     // increment
int incr2(int x);       // increment

int v = 2;
incr1(v);       // v becomes 3
v = incr2(v);   // v becomes 4
```

I think that for a reader, incr2() is easier to understand. That is, incr1() is more likely to lead to mistakes and errors. So, I'd prefer the style that returns a new value over the one that modifies a value as long as the creation and copy of a new value isn't expensive.

I do want to change the argument, should I use a pointer or should I use a reference? I don't know a strong logical reason. If passing ``not an object'' (e.g. a null pointer) is acceptable, using a pointer makes sense. My personal style is to use a pointer when I want to modify an object because in some contexts that makes it easier to spot that a modification is possible.

Note also that a call of a member function is essentially a call-by-reference on the object, so we often use member functions when we want to modify the value/state of an object.

Why is "this" not a reference?

Because "this" was introduced into C++ (really into C with Classes) before references were added. Also, I chose "this" to follow Simula usage, rather than the (later) Smalltalk use of "self".

What's wrong with arrays?

In terms of time and space, an array is just about the optimal construct for accessing a sequence of objects in memory. It is, however, also a very low level data structure with a vast potential for misuse and errors and in essentially all cases there are better alternatives. By "better" I mean easier to write, easier to read, less error prone, and as fast.

The two fundamental problems with arrays are that

- an array doesn't know its own size
- the name of an array converts to a pointer to its first element at the slightest provocation

Consider some examples:

```
void f(int a[], int s)
{
        // do something with a; the size of a is s
        for (int i = 0; i<s; ++i) a[i] = i;
}

int arr1[20];
int arr2[10];
```

```
void g()
{
        f(arr1,20);
        f(arr2,20);
}
```

The second call will scribble all over memory that doesn't belong to arr2. Naturally, a programmer usually get the size right, but it's extra work and ever so often someone makes the mistake. I prefer the simpler and cleaner version using the standard library vector:

```
void f(vector<int>& v)
{
        // do something with v
        for (int i = 0; i<v.size(); ++i) v[i] = i;
}

vector<int> v1(20);
vector<int> v2(10);

void g()
{
        f(v1);
        f(v2);
}
```

Since an array doesn't know its size, there can be no array assignment:

```
void f(int a[], int b[], int size)
{
        a = b;  // not array assignment
        memcpy(a,b,size);       // a = b
        // ...
}
```

Again, I prefer vector:

```
void g(vector<int>& a, vector<int>& b, int size)
{
        a = b;
        // ...
}
```

Another advantage of vector here is that memcpy() is not going to do the right thing for elements with copy constructors, such as strings.

```
void f(string a[], string b[], int size)
{
        a = b;  // not array assignment
        memcpy(a,b,size);       // disaster
        // ...
}

void g(vector<string>& a, vector<string>& b, int size)
{
        a = b;
        // ...
}
```

An array is of a fixed size determined at compile time:

```
const int S = 10;

void f(int s)
{
        int a1[s];      // error
        int a2[S];      // ok

        // if I want to extend a2, I'll have to change to an array
        // allocated on free store using malloc() and use realloc()
        // ...
}
```

To contrast:

```
const int S = 10;

void g(int s)
{
        vector<int> v1(s);      // ok
        vector<int> v2(S);      // ok
        v2.resize(v2.size()*2);
        // ...
}
```

C99 allows variable array bounds for local arrays, but those VLAs have their own problems.

The way that array names "decay" into pointers is fundamental to their use in C and C++. However, array decay interact very badly with inheritance. Consider:

```
class Base { void fct(); /* ... */ };
class Derived : Base { /* ... */ };

void f(Base* p, int sz)
{
        for (int i=0; i<sz; ++i) p[i].fct();
}

Base ab[20];
Derived ad[20];

void g()
{
        f(ab,20);
        f(ad,20);       // disaster!
}
```

In the last call, the Derived[] is treated as a Base[] and the subscripting no longer works correctly when sizeof(Derived)!=sizeof(Base) -- as will be the case in most cases of interest. If we used vectors instead, the error would be caught at compile time:

```
void f(vector<Base>& v)
{
        for (int i=0; i<v.size(); ++i) v[i].fct();
}

vector<Base> ab(20);
vector<Derived> ad(20);

void g()
{
        f(ab);
        f(ad);  // error: cannot convert a vector<Derived> to a vector<Base>
}
```

I find that an astonishing number of novice programming errors in C and C++ relate to (mis)uses of arrays.

Why doesn't C++ have a final keyword?

It has, but it is not as useful as you might think.

Should I use NULL or 0?

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

If you have to name the null pointer, call it nullptr; that's what it's called in C++11. Then, "nullptr" will be a keyword.

How are C++ objects laid out in memory?

Like C, C++ doesn't define layouts, just semantic constraints that must be met. Therefore different implementations do things differently. Unfortunately, the best explanation I know of is in a book that is otherwise outdated and doesn't describe any current C++ implementation: The Annotated C++ Reference Manual (usually called the ARM). It has diagrams of key layout examples. There is a very brief explanation in Chapter 2 of TC++PL.

Basically, C++ constructs objects simply by concatenating sub objects. Thus

```
struct A { int a,b; };
```

is represented by two ints next to each other, and

```
struct B : A { int c; };
```

is represented by an A followed by an int; that is, by three ints next to each other.

Virtual functions are typically implemented by adding a pointer (the vptr) to each object of a class with virtual functions. This pointer points to the appropriate table of functions (the vtbl). Each class has its own vtbl shared by all objects of that class.

What's the value of i++ + i++?

It's undefined. Basically, in C and C++, if you read a variable twice in an expression where you also write it, the result is undefined. Don't do that. Another example is:

```
v[i] = i++;
```

Related example:

```
f(v[i],i++);
```

Here, the result is undefined because the order of evaluation of function arguments are undefined.

Having the order of evaluation undefined is claimed to yield better performing code. Compilers could warn about such examples, which are typically subtle bugs (or potential subtle bugs). I'm disappointed that after decades, most compilers still don't warn, leaving that job to specialized, separate, and underused tools.

Why are some things left undefined in C++?

Because machines differ and because C left many things undefined. For details, including definitions of the terms "undefined", "unspecified", "implementation defined", and "well-formed"; see the ISO C++ standard. Note that the meaning of those terms differ from their definition of the ISO C standard and from some common usage. You can get wonderfully confused discussions when people don't realize that not everybody share definitions.

This is a correct, if unsatisfactory, answer. Like C, C++ is meant to exploit hardware directly and efficiently. This implies that C++ must deal with hardware entities such as bits, bytes, words, addresses, integer computations, and floating-point computations the way they are on a given machine, rather than how we might like them to be. Note that many "things" that people refer to as "undefined" are in fact "implementation defined", so that we can write perfectly specified code as long as we know which machine we are running on. Sizes of integers and the rounding behaviour of floating-point computations fall into that category.

Consider what is probably the the best known and most infamous example of undefined behavior:

```
int a[10];
a[100] = 0;     // range error
int* p = a;
// ...
```

```
p[100] = 0;     // range error (unless we gave p a better value before that assignment)
```

The C++ (and C) notion of array and pointer are direct representations of a machine's notion of memory and addresses, provided with no overhead. The primitive operations on pointers map directly onto machine instructions. In particular, no range checking is done. Doing range checking would impose a cost in terms of run time and code size. C was designed to outcompete assembly code for operating systems tasks, so that was a necessary decision. Also, C -- unlike C++ -- has no reasonable way of reporting a violation had a compiler decided to generate code to detect it: There are no exceptions in C. C++ followed C for reasons of compatibility and because C++ also compete directly with assembler (in OS, embedded systems, and some numeric computation areas). If you want range checking, use a suitable checked class (vector, smart pointer, string, etc.). A good compiler could catch the range error for a[100] at compile time, catching the one for p[100] is far more difficult, and in general it is impossible to catch every range error at compile time.

Other examples of undefined behavior stems from the compilation model. A compiler cannot detect an inconsistent definition of an object or a function in separately-compiled translation units. For example:

```
// file1.c:
struct S { int x,y; };
int f(struct S* p) { return p->x; }

// file2.c:
struct S { int y,x; }
int main()
{
        struct S s;
        s.x = 1;
        int x = f(&s);  // x!=s.x !!
        return 2;
}
```

Compiling file1.c and file2.c and linking the results into the same program is illegal in both C and C++. A linker could catch the inconsistent definition of S, but is not obliged to do so (and most don't). In many cases, it can be quite difficult to catch inconsistencies between separately compiled translation units. Consistent use of header files helps minimize such problems and there are some signs that linkers are improving. Note that C++ linkers do catch almost all errors related to inconsistently declared functions.

Finally, we have the apparently unnecessary and rather annoying undefined behavior of individual expressions. For example:

```
void out1() { cout << 1; }
void out2() { cout << 2; }

int main()
{
        int i = 10;
        int j = ++i + i++;      // value of j unspecified
        f(out1(),out2());       // prints 12 or 21
}
```

The value of j is unspecified to allow compilers to produce optimal code. It is claimed that the difference between what can be produced giving the compiler this freedom and requiring "ordinary left-to-right evaluation" can be significant. I'm unconvinced, but with innumerable compilers "out there" taking advantage of the freedom and some people passionately defending that freedom, a change would be difficult and could take decades to penetrate to the distant corners of the C and C++ worlds. I am disappointed that not all compilers warn against code such as ++i+i++. Similarly, the order of evaluation of arguments is unspecified.

IMO far too many "things" are left undefined, unspecified, implementation-defined, etc. However, that's easy to say and even to give examples of, but hard to fix. It should also be noted that it is not all that difficult to avoid most of the problems and produce portable code.

Why can't I define constraints for my template parameters?

Well, you can, and it's quite easy and general.

Consider:

```
template<class Container>
void draw_all(Container& c)
{
        for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

If there is a type error, it will be in the resolution of the fairly complicated for_each() call. For example, if the element type of the container is an int, then we get some kind of obscure error related to the for_each() call (because we can't invoke Shape::draw() for an int).

To catch such errors early, I can write:

```
template<class Container>
void draw_all(Container& c)
{
        Shape* p = c.front(); // accept only containers of Shape*s

        for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

The initialization of the spurious variable "p" will trigger a comprehensible error message from most current compilers. Tricks like this are common in all languages and have to be developed for all novel constructs. In production code, I'd probably write something like:

```
template<class Container>
void draw_all(Container& c)
{
        typedef typename Container::value_type T;
        Can_copy<T,Shape*>(); // accept containers of only Shape*s

        for_each(c.begin(),c.end(),mem_fun(&Shape::draw));
}
```

This makes it clear that I'm making an assertion. The Can_copy template can be defined like this:

```
template<class T1, class T2> struct Can_copy {
        static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
        Can_copy() { void(*p)(T1,T2) = constraints; }
};
```

Can_copy checks (at compile time) that a T1 can be assigned to a T2. Can_copy<T,Shape*> checks that T is a Shape* or a pointer to a class publicly derived from Shape or a type with a user-defined conversion to Shape*. Note that the definition is close to minimal:

- one line to name the constraints to be checked and the types for which to check them
- one line to list the specific constraints checked (the constraints() function)
- one line to provide a way to trigger the check (the constructor)

Note also that the definition has the desirable properties that

- You can express constraints without declaring or copying variables, thus the writer of a constraint doesn't have to make assumptions about how a type is initialized, whether objects can be copied, destroyed, etc. (unless, of course, those are the properties being tested by the constraint)
- No code is generated for a constraint using current compilers
- No macros are needed to define or use constraints
- Current compilers give acceptable error messages for a failed constraint, including the word "constraints" (to give the reader a clue), the name of the constraints, and the specific error that caused the failure (e.g. "cannot initialize Shape* by double*")

So why is something like Can_copy() - or something even more elegant - not in the language? D&E contains an analysis of the difficulties involved in expressing general constraints for C++. Since

then, many ideas have emerged for making these constraints classes easier to write and still trigger good error messages. For example, I believe the use of a pointer to function the way I do in Can_copy originates with Alex Stepanov and Jeremy Siek. I don't think that Can_copy() is quite ready for standardization - it needs more use. Also, different forms of constraints are in use in the C++ community; there is not yet a consensus on exactly what form of constraints templates is the most effective over a wide range of uses.

However, the idea is very general, more general than language facilities that have been proposed and provided specifically for constraints checking. After all, when we write a template we have the full expressive power of C++ available. Consider:

```
template<class T, class B> struct Derived_from {
        static void constraints(T* p) { B* pb = p; }
        Derived_from() { void(*p)(T*) = constraints; }
};

template<class T1, class T2> struct Can_copy {
        static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
        Can_copy() { void(*p)(T1,T2) = constraints; }
};

template<class T1, class T2 = T1> struct Can_compare {
        static void constraints(T1 a, T2 b) { a==b; a!=b; a<b; }
        Can_compare() { void(*p)(T1,T2) = constraints; }
};

template<class T1, class T2, class T3 = T1> struct Can_multiply {
        static void constraints(T1 a, T2 b, T3 c) { c = a*b; }
        Can_multiply() { void(*p)(T1,T2,T3) = constraints; }
};

struct B { };
struct D : B { };
struct DD : D { };
struct X { };

int main()
{
        Derived_from<D,B>();
        Derived_from<DD,B>();
        Derived_from<X,B>();
        Derived_from<int,B>();
        Derived_from<X,int>();

        Can_compare<int,float>();
        Can_compare<X,B>();
        Can_multiply<int,float>();
        Can_multiply<int,float,double>();
        Can_multiply<B,X>();

        Can_copy<D*,B*>();
        Can_copy<D,B*>();
        Can_copy<int,B*>();
}

// the classical "elements must derived from Mybase*" constraint:

template<class T> class Container : Derived_from<T,Mybase> {
        // ...
};
```

Actually, Derived_from doesn't check derivation, but conversion, but that's often a better constraint. Finding good names for constraints can be hard.

Why use sort() when we have "good old qsort()"?

To a novice,

```
qsort(array,asize,sizeof(elem),elem_compare);
```

looks pretty weird, and is harder to understand than

```
sort(vec.begin(),vec.end());
```

To an expert, the fact that sort() tends to be faster than qsort() for the same elements and the same comparison criteria is often significant. Also, sort() is generic, so that it can be used for any reasonable combination of container type, element type, and comparison criterion. For example:

```
struct Record {
        string name;
        // ...
};

struct name_compare {   // compare Records using "name" as the key
        bool operator()(const Record& a, const Record& b) const
                { return a.name<b.name; }
};

void f(vector<Record>& vs)
{
        sort(vs.begin(), vs.end(), name_compare());
        // ...
}
```

In addition, most people appreciate that sort() is type safe, that no casts are required to use it, and that they don't have to write a compare() function for standard types.

For a more detailed explanation, see my paper "Learning C++ as a New language", which you can download from my publications list.

The primary reason that sort() tends to outperform qsort() is that the comparison inlines better.

What is a function object?

An object that in some way behaves like a function, of course. Typically, that would mean an object of a class that defines the application operator - operator().

A function object is a more general concept than a function because a function object can have state that persist across several calls (like a static local variable) and can be initialized and examined from outside the object (unlike a static local variable). For example:

```
class Sum {
        int val;
public:
        Sum(int i) :val(i) { }
        operator int() const { return val; }            // extract value

        int operator()(int i) { return val+=i; }        // application
};

void f(vector<int> v)
{
        Sum s = 0;      // initial value 0
        s = for_each(v.begin(), v.end(), s);    // gather the sum of all elements
        cout << "the sum is " << s << "\n";

        // or even:
        cout << "the sum is " << for_each(v.begin(), v.end(), Sum(0)) << "\n";
}
```

Note that a function object with an inline application operator inlines beautifully because there are no pointers involved that might confuse optimizers. To contrast: current optimizers are rarely (never?) able to inline a call through a pointer to function.

Function objects are extensively used to provide flexibility in the standard library.

How do I deal with memory leaks?

By writing code that doesn't have any.

Clearly, if your code has new operations, delete operations, and pointer arithmetic all over the place, you are going to mess up somewhere and get leaks, stray pointers, etc. This is true independently of how conscientious you are with your allocations: eventually the complexity of the code will overcome the time and effort you can afford. It follows that successful techniques rely on hiding allocation and deallocation inside more manageable types. Good examples are the standard containers. They manage memory for their elements better than you could without disproportionate effort. Consider writing this without the help of string and vector:

```
#include<vector>
#include<string>
#include<iostream>
#include<algorithm>
using namespace std;

int main()      // small program messing around with strings
{
        cout << "enter some whitespace-separated words:\n";
        vector<string> v;
        string s;
        while (cin>>s) v.push_back(s);

        sort(v.begin(),v.end());

        string cat;
        typedef vector<string>::const_iterator Iter;
        for (Iter p = v.begin(); p!=v.end(); ++p) cat += *p+"+";
        cout << cat << '\n';
}
```

What would be your chance of getting it right the first time? And how would you know you didn't have a leak?

Note the absence of explicit memory management, macros, casts, overflow checks, explicit size limits, and pointers. By using a function object and a standard algorithm, I could have eliminated the pointer-like use of the iterator, but that seemed overkill for such a tiny program.

These techniques are not perfect and it is not always easy to use them systematically. However, they apply surprisingly widely and by reducing the number of explicit allocations and deallocations you make the remaining examples much easier to keep track of. As early as 1981, I pointed out that by reducing the number of objects that I had to keep track of explicitly from many tens of thousands to a few dozens, I had reduced the intellectual effort needed to get the program right from a Herculean task to something manageable, or even easy.

If your application area doesn't have libraries that make programming that minimizes explicit memory management easy, then the fastest way of getting your program complete and correct might be to first build such a library.

Templates and the standard libraries make this use of containers, resource handles, etc., much easier than it was even a few years ago. The use of exceptions makes it close to essential.

If you cannot handle allocation/deallocation implicitly as part of an object you need in your application anyway, you can use a resource handle to minimize the chance of a leak. Here is an example where I need to return an object allocated on the free store from a function. This is an opportunity to forget to delete that object. After all, we cannot tell just looking at pointer whether it needs to be deallocated and if so who is responsible for that. Using a resource handle, here the standard library auto_ptr, makes it clear where the responsibility lies:

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

struct S {
```

```
        S() { cout << "make an S\n"; }
        ~S() { cout << "destroy an S\n"; }
        S(const S&) { cout << "copy initialize an S\n"; }
        S& operator=(const S&) { cout << "copy assign an S\n"; }
};

S* f()
{
        return new S;   // who is responsible for deleting this S?
};

auto_ptr<S> g()
{
        return auto_ptr<S>(new S);      // explicitly transfer responsibility for deleting this S
}

int main()
{
        cout << "start main\n";
        S* p = f();
        cout << "after f() before g()\n";
//      S* q = g();     // this error would be caught by the compiler
        auto_ptr<S> q = g();
        cout << "exit main\n";
        // leaks *p
        // implicitly deletes *q
}
```

Think about resources in general, rather than simply about memory.

If systematic application of these techniques is not possible in your environment (you have to use code from elsewhere, part of your program was written by Neanderthals, etc.), be sure to use a memory leak detector as part of your standard development procedure, or plug in a garbage collector.

Why can't I resume after catching an exception?

In other words, why doesn't C++ provide a primitive for returning to the point from which an exception was thrown and continuing execution from there?

Basically, someone resuming from an exception handler can never be sure that the code after the point of throw was written to deal with the execution just continuing as if nothing had happened. An exception handler cannot know how much context to "get right" before resuming. To get such code right, the writer of the throw and the writer of the catch need intimate knowledge of each others code and context. This creates a complicated mutual dependency that wherever it has been allowed has led to serious maintenance problems.

I seriously considered the possibility of allowing resumption when I designed the C++ exception handling mechanism and this issue was discussed in quite some detail during standardization. See the exception handling chapter of The Design and Evolution of C++.

If you want to check to see if you can fix a problem before throwing an exception, call a function that checks and then throws only if the problem cannot be dealt with locally. A new_handler is an example of this.

Why doesn't C++ have an equivalent to realloc()?

If you want to, you can of course use realloc(). However, realloc() is only guaranteed to work on arrays allocated by malloc() (and similar functions) containing objects without user-defined copy constructors. Also, please remember that contrary to naive expectations, realloc() occasionally does copy its argument array.

In C++, a better way of dealing with reallocation is to use a standard library container, such as

vector, and let it grow naturally.

Why use exceptions?

What good can using exceptions do for me? The basic answer is: Using exceptions for error handling makes you code simpler, cleaner, and less likely to miss errors. But what's wrong with "good old **errno** and **if**-statements"? The basic answer is: Using those, your error handling and your normal code are closely intertwined. That way, your code gets messy and it becomes hard to ensure that you have dealt with all errors (think "spaghetti code" or a "rat's nest of tests").

First of all there are things that just can't be done right without exceptions. Consider an error detected in a constructor; how do you report the error? You throw an exception. That's the basis of RAII (Resource Acquisition Is Initialization), which it the basis of some of the most effective modern C++ design techniques: A constructor's job is to establish the invariant for the class (create the environment in which the members function are to run) and that often requires the acquisition of resources, such as memory, locks, files, sockets, etc.

Imagine that we did not have exceptions, how would you deal with an error detected in a constructor? Remember that constructors are often invoked initialize/construct objects in variables:

```
vector<double> v(100000);   // needs to allocate memory
ofstream os("myfile");      // needs to open a file
```

The **vector** or **ofstream** (output file stream) constructor could either set the variable into a "bad" state (as ifstream does by default) so that every subsequent operation fails. That's *not* ideal. For example, in the case of **ofstream**, your output simply disappears if you forget to check that the open operation succeeded. For most classes that results are worse. At least, we would have to write:

```
vector<double> v(100000);   // needs to allocate memory
if (v.bad()) { /* handle error */ }     // vector doesn't actually have a bad(); it relies on exceptions
ofstream os("myfile");      // needs to open a file
if (os.bad())  { /* handle error */ }
```

That's an extra test per object (to write, to remember or forget). This gets really messy for classes composed of several objects, especially if those sub-objects depend on each other. For more information see The C++ Programming Language section 8.3, Chapter 14, and Appendix E or the (more academic) paper Exception safety: Concepts and techniques.

So writing constructors can be tricky without exceptions, but what about plain old functions? We can either return an error code or set a non-local variable (e.g. **errno**). Setting a global variable doesn't work too well unless you test it immediately (or some other function might have re-set it). Don't even think of that technique if you might have multiple threads accessing the global variable. The trouble with return values are that choosing the error return value can require cleverness and can be impossible:

```
double d = my_sqrt(-1);         // return -1 in case of error
if (d == -1) { /* handle error */ }
int x = my_negate(INT_MIN);     // Duh?
```

There is no possible value for **my_negate()** to return: Every possible **int** is the correct answer for some **int** and there is no correct answer for the most negative number in the twos-complement representation. In such cases, we would need to return pairs of values (and as usual remember to test) See my Beginning programming book for more examples and explanations.

Common objections to the use of exceptions:

- *but exceptions are expensive!*: Not really. Modern C++ implementations reduce the overhead of using exceptions to a few percent (say, 3%) and that's compared to no error handling. Writing code with error-return codes and tests is not free either. As a rule of thumb, exception handling is extremely cheap when you don't throw an exception. It costs nothing on some implementations. All the cost is incurred when you throw an exception: that is, "normal

code" is faster than code using error-return codes and tests. You incur cost only when you have an error.
- *but in JSF++ you yourself ban exceptions outright!*: JSF++ is for hard-real time and safety-critical applications (flight control software). If a computation takes too long someone may die. For that reason, we have to *guarantee* response times, and we can't - with the current level of tool support - do that for exceptions. In that context, even free store allocation is banned! Actually, the JSF++ recommendations for error handling simulate the use of exceptions in anticipation of the day where we have the tools to do things right, i.e. using exceptions.
- *but throwing an exception from a constructor invoked by **new** causes a memory leak!*: Nonsense! That's an old-wives' tale caused by a bug in one compiler - and that bug was immediately fixed over a decade ago.

How do I use exceptions?

See The C++ Programming Language section 8.3, Chapter 14, and Appendix E. The appendix focuses on techniques for writing exception-safe code in demanding applications, and is not written for novices.

In C++, exceptions is used to signal errors that cannot be handled locally, such as the failure to acquire a resource in a constructor. For example:

```
class Vector {
        int sz;
        int* elem;
        class Range_error { };
public:
        Vector(int s) : sz(s) { if (sz<0) throw Range_error(); /* ... */ }
        // ...
};
```

Do not use exceptions as simply another way to return a value from a function. Most users assume - as the language definition encourages them to - that exception-handling code is error-handling code, and implementations are optimized to reflect that assumption.

A key technique is resource acquisiton is initialization (sometimes abbreviated to RAII), which uses classes with destructors to impose order on resource management. For example:

```
void fct(string s)
{
        File_handle f(s,"r");   // File_handle's constructor opens the file called "s"
        // use f
} // here File_handle's destructor closes the file
```

If the "use f" part of fct() throws an exception, the destructor is still invoked and the file is properly closed. This contrasts to the common unsafe usage:

```
void old_fct(const char* s)
{
        FILE* f = fopen(s,"r"); // open the file named "s"
        // use f
        fclose(f);      // close the file
}
```

If the "use f" part of old_fct throws an exception - or simply does a return - the file isn't closed. In C programs, longjmp() is an additional hazard.

Why can't I assign a vector<Apple*> to a vector<Fruit*>?

Because that would open a hole in the type system. For example:

```
class Apple : public Fruit { void apple_fct(); /* ... */ };
class Orange : public Fruit { /* ... */ }; // Orange doesn't have apple_fct()
```

```
vector<Apple*> v;       // vector of Apples

void f(vector<Fruit*>& vf)              // innocent Fruit manipulating function
{
        vf.push_back(new Orange);       // add orange to vector of fruit
}

void h()
{
        f(v);   // error: cannot pass a vector<Apple*> as a vector<Fruit*>
        for (int i=0; i<v.size(); ++i) v[i]->apple_fct();
}
```

Had the call f(v) been legal, we would have had an Orange pretending to be an Apple.

An alternative language design decision would have been to allow the unsafe conversion, but rely on dynamic checking. That would have required a run-time check for each access to v's members, and h() would have had to throw an exception upon encountering the last element of v.

Why doesn't C++ have a universal class Object?

- We don't need one: generic programming provides statically type safe alternatives in most cases. Other cases are handled using multiple inheritance.
- There is no useful universal class: a truly universal carries no semantics of its own.
- A "universal" class encourages sloppy thinking about types and interfaces and leads to excess run-time checking.
- Using a universal base class implies cost: Objects must be heap-allocated to be polymorphic; that implies memory and access cost. Heap objects don't naturally support copy semantics. Heap objects don't support simple scoped behavior (which complicates resource management). A universal base class encourages use of dynamic_cast and other run-time checking.

Yes. I have simplified the arguments; this is an FAQ, not an academic paper.

Do we really need multiple inheritance?

Not really. We can do without multiple inheritance by using workarounds, exactly as we can do without single inheritance by using workarounds. We can even do without classes by using workarounds. C is a proof of that contention. However, every modern language with static type checking and inheritance provides some form of multiple inheritance. In C++, abstract classes often serve as interfaces and a class can have many interfaces. Other languages - often deemed "not MI" - simply has a separate name for their equivalent to a pure abstract class: an interface. The reason languages provide inheritance (both single and multiple) is that language-supported inheritance is typically superior to workarounds (e.g. use of forwarding functions to sub-objects or separately allocated objects) for ease of programming, for detecting logical problems, for maintainability, and often for performance.

How do I read a string from input?

You can read a single, whitespace terminated word like this:

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

int main()
{
        cout << "Please enter a word:\n";

        string s;
        cin>>s;
```

```
        cout << "You entered " << s << '\n';
}
```

Note that there is no explicit memory management and no fixed-sized buffer that you could possibly overflow.

If you really need a whole line (and not just a single word) you can do this:

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

int main()
{
        cout << "Please enter a line:\n";

        string s;
        getline(cin,s);

        cout << "You entered " << s << '\n';
}
```

For a brief introduction to standard library facilities, such as iostream and string, see Chaper 3 of TC++PL3 (available online). For a detailed comparison of simple uses of C and C++ I/O, see "Learning Standard C++ as a New Language", which you can download from my publications list

---

Is "generics" what templates should have been?

No. generics are primarily syntactic sugar for abstract classes; that is, with generics (whether Java or C# generics), you program against precisely defined interfaces and typically pay the cost of virtual function calls and/or dynamic casts to use arguments.

Templates supports generic programming, template metaprogramming, etc. through a combination of features such as integer template arguments, specialization, and uniform treatment of built-in and user-defined types. The result is flexibility, generality, and performance unmatched by "generics". The STL is the prime example.

A less desirable result of the flexibility is late detection of errors and horrendously bad error messages. This is currently being addressed indirectly with constraints classes.

Can I throw an exception from a constructor? From a destructor?

- Yes: You should throw an exception from a constructor whenever you cannot properly initialize (construct) an object. There is no really satisfactory alternative to exiting a constructor by a throw.
- Not really: You can throw an exception in a destructor, but that exception must not leave the destructor; if a destructor exits by a throw, all kinds of bad things are likely to happen because the basic rules of the standard library and the language itself will be violated. Don't do it.

For examples and detailed explanations, see Appendix E of The C++ Programming Language.

There is a caveat: Exceptions can't be used for some hard-real time projects. For example, see the JSF air vehicle C++ coding standards.

Why doesn't C++ provide a "finally" construct?

Because C++ supports an alternative that is almost always better: The "resource acquisition is

initialization" technique (TC++PL3 section 14.4). The basic idea is to represent a resource by a local object, so that the local object's destructor will release the resource. That way, the programmer cannot forget to release the resource. For example:

```
class File_handle {
        FILE* p;
public:
        File_handle(const char* n, const char* a)
                { p = fopen(n,a); if (p==0) throw Open_error(errno); }
        File_handle(FILE* pp)
                { p = pp; if (p==0) throw Open_error(errno); }

        ~File_handle() { fclose(p); }

        operator FILE*() { return p; }

        // ...
};

void f(const char* fn)
{
        File_handle f(fn,"rw"); // open fn for reading and writing
        // use file through f
}
```

In a system, we need a "resource handle" class for each resource. However, we don't have to have an "finally" clause for each acquisition of a resource. In realistic systems, there are far more resource acquisitions than kinds of resources, so the "resource acquisition is initialization" technique leads to less code than use of a "finally" construct.

Also, have a look at the examples of resource management in Appendix E of The C++ Programming Language.

What is an auto_ptr and why isn't there an auto_array?

An auto_ptr is an example of very simple handle class, defined in <memory>, supporting exception safety using the resource acquisition is initialization technique. An auto_ptr holds a pointer, can be used as a pointer, and deletes the object pointed to at the end of its scope. For example:

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

struct X {
        int m;
        // ..
};

void f()
{
        auto_ptr<X> p(new X);
        X* q = new X;

        p->m++;         // use p just like a pointer
        q->m++;
        // ...

        delete q;
}
```

If an exception is thrown in the ... part, the object held by p is correctly deleted by auto_ptr's destructor while the X pointed to by q is leaked. See TC++PL 14.4.2 for details.

Auto_ptr is a very lightweight class. In particular, it is *not* a reference counted pointer. If you "copy" one auto_ptr into another, the assigned to auto_ptr holds the pointer and the assigned auto_ptr holds 0. For example:

```
#include<memory>
```

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

struct X {
        int m;
        // ..
};

int main()
{
        auto_ptr<X> p(new X);
        auto_ptr<X> q(p);
        cout << "p " << p.get() << " q " << q.get() << "\n";
}
```

should print a 0 pointer followed by a non-0 pointer. For example:

```
p 0x0 q 0x378d0
```

auto_ptr::get() returns the held pointer.

This "move semantics" differs from the usual "copy semantics", and can be surprising. In particular, never use an **auto_ptr** as a member of a standard container. The standard containers require the usual copy semantics. For example:

```
std::vector<auto_ptr<X> >v;     // error
```

An auto_ptr holds a pointer to an individual element, not a pointer to an array:

```
void f(int n)
{
        auto_ptr<X> p(new X[n]);        // error
        // ...
}
```

This is an error because the destructor will delete the pointer using **delete** rather than **delete[]** and will fail to invoke the destructor for the last n-1 Xs.

So should we use an auto_array to hold arrays? No. There is no auto_array. The reason is that there isn't a need for one. A better solution is to use a **vector**:

```
void f(int n)
{
        vector<X> v(n);
        // ...
}
```

Should an exception occur in the ... part, **v**'s destructor will be correctly invoked.

In C++11 use a Unique_ptr instead of **auto_ptr**.

What shouldn't I use exceptions for?

C++ exceptions are designed to support error handling. Use **throw** only to signal an error and **catch** only to specify error handling actions. There are other uses of exceptions - popular in other languages - but not idiomatic in C++ and deliberately not supported well by C++ implementations (those implementations are optimized based on the assumption that exceptions are used for error handling).

In particular, **throw** is not simply an alternative way of returning a value from a function (similar to **return**). Doing so will be slow and will confuse most C++ programmers used to seing exceptions used only for error handling. Similarly, **throw** is *not* a good way of getting out of a loop.

What is the difference between new and malloc()?

malloc() is a function that takes a number (of bytes) as its argument; it returns a **void*** pointing to unitialized storage. **new** is an operator that takes a type and (optionally) a set of initializers for that type as its arguments; it returns a pointer to an (optionally) initialized object of its type. The difference is most obvious when you want to allocate an object of a user-defined type with non-trivial initialization semantics. Examples:

```
class Circle : public Shape {
public:
        Cicle(Point c, int r);
        // no default constructor
        // ...
};

class X {
public:
        X();    // default constructor
        // ...
};

void f(int n)
{
        void* p1 = malloc(40);  // allocate 40 (uninitialized) bytes

        int* p2 = new int[10];  // allocate 10 uninitialized ints
        int* p3 = new int(10);  // allocate 1 int initialized to 10
        int* p4 = new int();    // allocate 1 int initialized to 0
        int* p4 = new int;      // allocate 1 uninitialized int

        Circle* pc1 = new Circle(Point(0,0),10); // allocate a Circle constructed
                                                 // with the specified argument
        Circle* pc2 = new Circle;       // error no default constructor

        X* px1 = new X;         // allocate a default constructed X
        X* px2 = new X();       // allocate a default constructed X
        X* px2 = new X[10];     // allocate 10 default constructed Xs
        // ...
}
```

Note that when you specify a initializer using the "(value)" notation, you get initialization with that value. Unfortunately, you cannot specify that for an array. Often, a **vector** is a better alternative to a free-store-allocated array (e.g., consider exception safety).

Whenever you use **malloc()** you must consider initialization and convertion of the return pointer to a proper type. You will also have to consider if you got the number of bytes right for your use. There is no performance difference between **malloc()** and **new** when you take initialization into account.

malloc() reports memory exhaustion by returning 0. **new** reports allocation and initialization errors by throwing exceptions.

Objects created by **new** are destroyed by **delete**. Areas of memory allocated by **malloc()** are deallocated by **free()**.

Can I mix C-style and C++ style allocation and deallocation?

Yes, in the sense that you can use malloc() and new in the same program.

No, in the sense that you cannot allocate an object with malloc() and free it using delete. Nor can you allocate with new and delete with free() or use realloc() on an array allocated by new.

The C++ operators new and delete guarantee proper construction and destruction; where constructors or destructors need to be invoked, they are. The C-style functions malloc(), calloc(), free(), and realloc() doesn't ensure that. Furthermore, there is no guarantee that the mechanism used by new and delete to acquire and release raw memory is compatible with malloc() and free(). If mixing styles works on your system, you were simply "lucky" - for now.

If you feel the need for realloc() - and many do - then consider using a standard library vector. For example

```
// read words from input into a vector of strings:

vector<string> words;
string s;
while (cin>>s && s!=".") words.push_back(s);
```

The vector expands as needed.

See also the examples and discussion in "Learning Standard C++ as a New Language", which you can download from my publications list.

Why must I use a cast to convert from void*?

In C, you can implicitly convert a void* to a T*. This is unsafe. Consider:

```
#include<stdio.h>

int main()
{
        char i = 0;
        char j = 0;
        char* p = &i;
        void* q = p;
        int* pp = q;    /* unsafe, legal C, not C++ */

        printf("%d %d\n",i,j);
        *pp = -1;       /* overwrite memory starting at &i */
        printf("%d %d\n",i,j);
}
```

The effects of using a T* that doesn't point to a T can be disastrous. Consequently, in C++, to get a T* from a void* you need an explicit cast. For example, to get the undesirable effects of the program above, you have to write:

```
        int* pp = (int*)q;
```

or, using a new style cast to make the unchecked type conversion operation more visible:

```
        int* pp = static_cast<int*>(q);
```

Casts are best avoided.

One of the most common uses of this unsafe conversion in C is to assign the result of malloc() to a suitable pointer. For example:

```
int* p = malloc(sizeof(int));
```

In C++, use the typesafe new operator:

```
int* p = new int;
```

Incidentally, the new operator offers additional advantages over malloc():

- new can't accidentally allocate the wrong amount of memory,
- new implicitly checks for memory exhaustion, and
- new provides for initialization

For example:

```
typedef std::complex<double> cmplx;

/* C style: */
cmplx* p = (cmplx*)malloc(sizeof(int)); /* error: wrong size */
                                                    /* forgot to test for p==0 */
if (*p == 7) { /* ... */ }                          /* oops: forgot to initialize *p */

// C++ style:
```

```
        cmplx* q = new cmplx(1,2); // will throw bad_alloc if memory is exhausted
        if (*q == 7) { /* ... */ }
```

How do I define an in-class constant?

If you want a constant that you can use in a constant expression, say as an array bound, you have two choices:

```
class X {
        static const int c1 = 7;
        enum { c2 = 19 };

        char v1[c1];
        char v2[c2];

        // ...
};
```

At first glance, the declaration of c1 seems cleaner, but note that to use that in-class initialization syntax, the constant must be a static const of integral or enumeration type initialized by a constant expression. That's quite restrictive:

```
class Y {
        const int c3 = 7;               // error: not static
        static int c4 = 7;              // error: not const
        static const float c5 = 7;      // error: not integral
};
```

I tend to use the "enum trick" because it's portable and doesn't tempt me to use non-standard extensions of the in-class initialization syntax.

So why do these inconvenient restrictions exist? A class is typically declared in a header file and a header file is typically included into many translation units. However, to avoid complicated linker rules, C++ requires that every object has a unique definition. That rule would be broken if C++ allowed in-class definition of entities that needed to be stored in memory as objects. See D&E for an explanation of C++'s design tradeoffs.

You have more flexibility if the const isn't needed for use in a constant expression:

```
        class Z {
                static char* p;         // initialize in definition
                const int i;            // initialize in constructor
        public:
                Z(int ii) :i(ii) { }
        };

        char* Z::p = "hello, there";
```

You can take the address of a static member if (and only if) it has an out-of-class definition:

```
        class AE {
                // ...
        public:
                static const int c6 = 7;
                static const int c7 = 31;
        };

        const int AE::c7;       // definition

        int f()
        {
                const int* p1 = &AE::c6;        // error: c6 not an lvalue
                const int* p2 = &AE::c7;        // ok
                // ...
        }
```

Why doesn't delete zero out its operand?

Consider

```
delete p;
// ...
delete p;
```

If the ... part doesn't touch p then the second "delete p;" is a serious error that a C++ implementation cannot effectively protect itself against (without unusual precautions). Since deleting a zero pointer is harmless by definition, a simple solution would be for "delete p;" to do a "p=0;" after it has done whatever else is required. However, C++ doesn't guarantee that.

One reason is that the operand of delete need not be an lvalue. Consider:

```
delete p+1;
delete f(x);
```

Here, the implementation of delete does not have a pointer to which it can assign zero. These examples may be rare, but they do imply that it is not possible to guarantee that ``any pointer to a deleted object is 0.'' A simpler way of bypassing that ``rule'' is to have two pointers to an object:

```
T* p = new T;
T* q = p;
delete p;
delete q;       // ouch!
```

C++ explicitly allows an implementation of delete to zero out an lvalue operand, and I had hoped that implementations would do that, but that idea doesn't seem to have become popular with implementers.

If you consider zeroing out pointers important, consider using a destroy function:

```
template<class T> inline void destroy(T*& p) { delete p; p = 0; }
```

Consider this yet-another reason to minimize explicit use of new and delete by relying on standard library containers, handles, etc.

Note that passing the pointer as a reference (to allow the pointer to be zero'd out) has the added benefit of preventing destroy() from being called for an rvalue:

```
int* f();
int* p;
// ...
destroy(f());   // error: trying to pass an rvalue by non-const reference
destroy(p+1);   // error: trying to pass an rvalue by non-const reference
```

Why isn't the destructor called at the end of scope?

The simple answer is "of course it is!", but have a look at the kind of example that often accompany that question:

```
void f()
{
        X* p = new X;
        // use p
}
```

That is, there was some (mistaken) assumption that the object created by "new" would be destroyed at the end of a function.

Basically, you should only use "new" if you want an object to live beyond the lifetime of the scope you create it in. That done, you need to use "delete" to destroy it. For example:

```
X* g(int i) { /* ... */ return new X(i); }      // the X outlives the call of g()

void h(int i)
{
        X* p = g(i);
        // ...
```

```
        delete p;
}
```

If you want an object to live in a scope only, don't use "new" but simply define a variable:

```
{
        ClassName x;
        // use x
}
```

The variable is implicitly destroyed at the end of the scope.

Code that creates an object using new and then deletes it at the end of the same scope is ugly, error-prone, and inefficient. For example:

```
void fct()      // ugly, error-prone, and inefficient
{
        X* p = new X;
        // use p
        delete p;
}
```

Can I write "void main()"?

The definition

```
void main() { /* ... */ }
```

is not and never has been C++, nor has it even been C. See the ISO C++ standard 3.6.1[2] or the ISO C standard 5.1.2.2.1. A conforming implementation accepts

```
int main() { /* ... */ }
```

and

```
int main(int argc, char* argv[]) { /* ... */ }
```

A conforming implementation may provide more versions of main(), but they must all have return type int. The int returned by main() is a way for a program to return a value to "the system" that invokes it. On systems that doesn't provide such a facility the return value is ignored, but that doesn't make "void main()" legal C++ or legal C. Even if your compiler accepts "void main()" avoid it, or risk being considered ignorant by C and C++ programmers.

In C++, main() need not contain an explicit return statement. In that case, the value returned is 0, meaning successful execution. For example:

```
#include<iostream>

int main()
{
        std::cout << "This program returns the integer value 0\n";
}
```

Note also that neither ISO C++ nor C99 allows you to leave the type out of a declaration. That is, in contrast to C89 and ARM C++ ,"int" is not assumed where a type is missing in a declaration. Consequently:

```
#include<iostream>

main() { /* ... */ }
```

is an error because the return type of main() is missing.

Why can't I overload dot, ::, sizeof, etc.?

Most operators can be overloaded by a programmer. The exceptions are

```
. (dot)  ::  ?:  sizeof
```

There is no fundamental reason to disallow overloading of ?:. I just didn't see the need to introduce the special case of overloading a ternary operator. Note that a function overloading *expr1?expr2:expr3* would not be able to guarantee that only one of *expr2* and *expr3* was executed.

Sizeof cannot be overloaded because built-in operations, such as incrementing a pointer into an array implicitly depends on it. Consider:

```
X a[10];
X* p = &a[3];
X* q = &a[3];
p++;    // p points to a[4]
        // thus the integer value of p must be
        // sizeof(X) larger than the integer value of q
```

Thus, sizeof(X) could not be given a new and different meaning by the programmer without violating basic language rules.

In N::m neither N nor m are expressions with values; N and m are names known to the compiler and :: performs a (compile time) scope resolution rather than an expression evaluation. One could imagine allowing overloading of x::y where x is an object rather than a namespace or a class, but that would - contrary to first appearances - involve introducing new syntax (to allow *expr::expr*). It is not obvious what benefits such a complication would bring.

Operator . (dot) could in principle be overloaded using the same technique as used for ->. However, doing so can lead to questions about whether an operation is meant for the object overloading . or an object referred to by . For example:

```
class Y {
public:
        void f();
        // ...
};

class X {       // assume that you can overload .
        Y* p;
        Y& operator.() { return *p; }
        void f();
        // ...
};

void g(X& x)
{
        x.f();  // X::f or Y::f or error?
}
```

This problem can be solved in several ways. At the time of standardization, it was not obvious which way would be best. For more details, see D&E.

Can I define my own operators?

Sorry, no. The possibility has been considered several times, but each time I/we decided that the likely problems outweighed the likely benefits.

It's not a language-technical problem. Even when I first considerd it in 1983, I knew how it could be implemented. However, my experience has been that when we go beyond the most trivial examples people seem to have subtlely different opinions of "the obvious" meaning of uses of an operator. A classical example is a**b**c. Assume that ** has been made to mean exponentiation. Now should a**b**c mean (a**b)**c or a**(b**c)? I thought the answer was obvious and my friends agreed - and then we found that we didn't agree on which resolution was the obvious one. My conjecture is that such problems would lead to subtle bugs.

How do I convert an integer to a string?

The simplest way is to use a stringstream:

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

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

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

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

Naturally, this technique works for converting any type that you can output using << to a string. For a description of string streams, see 21.5.3 of The C++ Programming Language.

How do I call a C function from C++?

Just declare the C function ``extern "C"'' (in your C++ code) and call it (from your C or C++ code). For example:

```
// C++ code

extern "C" void f(int); // one way

extern "C" {    // another way
        int g(double);
        double h();
};

void code(int i, double d)
{
        f(i);
        int ii = g(d);
        double dd = h();
        // ...
}
```

The definitions of the functions may look like this:

```
/* C code: */

void f(int i)
{
        /* ... */
}

int g(double d)
{
        /* ... */
}

double h()
{
        /* ... */
}
```

Note that C++ type rules, not C rules, are used. So you can't call function declared ``extern "C"'' with the wrong number of argument. For example:

```
// C++ code
```

```
void more_code(int i, double d)
{
        double dd = h(i,d);     // error: unexpected arguments
        // ...
}
```

How do I call a C++ function from C?

Just declare the C++ function ``extern "C"'' (in your C++ code) and call it (from your C or C++ code). For example:

```
// C++ code:

extern "C" void f(int);

void f(int i)
{
        // ...
}
```

Now f() can be used like this:

```
/* C code: */

void f(int);

void cc(int i)
{
        f(i);
        /* ... */
}
```

Naturally, this works only for non-member functions. If you want to call member functions (incl. virtual functions) from C, you need to provide a simple wrapper. For example:

```
// C++ code:

class C {
        // ...
        virtual double f(int);
};

extern "C" double call_C_f(C* p, int i) // wrapper function
{
        return p->f(i);
}
```

Now C::f() can be used like this:

```
/* C code: */

double call_C_f(struct C* p, int i);

void ccc(struct C* p, int i)
{
        double d = call_C_f(p,i);
        /* ... */
}
```

If you want to call overloaded functions from C, you must provide wrappers with distinct names for the C code to use. For example:

```
// C++ code:

void f(int);
void f(double);

extern "C" void f_i(int i) { f(i); }
extern "C" void f_d(double d) { f(d); }
```

Now the f() functions can be used like this:

```
/* C code: */

void f_i(int);
void f_d(double);

void cccc(int i,double d)
{
        f_i(i);
        f_d(d);
        /* ... */
}
```

Note that these techniques can be used to call a C++ library from C code even if you cannot (or do not want to) modify the C++ headers.

Is ``int* p;'' right or is ``int *p;'' right?

Both are "right" in the sense that both are valid C and C++ and both have exactly the same meaning. As far as the language definitions and the compilers are concerned we could just as well say ``int*p;'' or ``int * p;''

The choice between ``int* p;'' and ``int *p;'' is not about right and wrong, but about style and emphasis. C emphasized expressions; declarations were often considered little more than a necessary evil. C++, on the other hand, has a heavy emphasis on types.

A ``typical C programmer'' writes ``int *p;'' and explains it ``*p is what is the int'' emphasizing syntax, and may point to the C (and C++) declaration grammar to argue for the correctness of the style. Indeed, the * binds to the name p in the grammar.

A ``typical C++ programmer'' writes ``int* p;'' and explains it ``p is a pointer to an int'' emphasizing type. Indeed the type of p is int*. I clearly prefer that emphasis and see it as important for using the more advanced parts of C++ well.

The critical confusion comes (only) when people try to declare several pointers with a single declaration:

```
int* p, p1;     // probable error: p1 is not an int*
```

Placing the * closer to the name does not make this kind of error significantly less likely.

```
int *p, p1;     // probable error?
```

Declaring one name per declaration minimizes the problem - in particular when we initialize the variables. People are far less likely to write:

```
int* p = &i;
int p1 = p;     // error: int initialized by int*
```

And if they do, the compiler will complain.

Whenever something can be done in two ways, someone will be confused. Whenever something is a matter of taste, discussions can drag on forever. Stick to one pointer per declaration and always initialize variables and the source of confusion disappears. See The Design and Evolution of C++ for a longer discussion of the C declaration syntax.

Which layout style is the best for my code?

Such style issues are a matter of personal taste. Often, opinions about code layout are strongly held, but probably consistency matters more than any particular style. Like most people, I'd have a hard time constructing a solid logical argument for my preferences.

I personally use what is often called "K&R" style. When you add conventions for constructs not found in C, that becomes what is sometimes called "Stroustrup" style. For example:

```
class C : public B {
public:
        // ...
};

void f(int* p, int max)
{
        if (p) {
                // ...
        }

        for (int i = 0; i<max; ++i) {
                // ...
        }
}
```

This style conserves vertical space better than most layout styles, and I like to fit as much as is reasonable onto a screen. Placing the opening brace of a function on a new line helps me distinguish function definition from class definitions at a glance.

Indentation is very important.

Design issues, such as the use of abstract classes for major interfaces, use of templates to present flexible type-safe abstractions, and proper use of exceptions to represent errors, are far more important than the choice of layout style.

How do you name variables? Do you recommend "Hungarian"?

No I don't recommend "Hungarian". I regard "Hungarian" (embedding an abbreviated version of a type in a variable name) a technique that can be useful in untyped languages, but is completely unsuitable for a language that supports generic programming and object-oriented programming - both of which emphasize selection of operations based on the type an arguments (known to the language or to the run-time support). In this case, "building the type of an object into names" simply complicates and minimizes abstraction. To various extent, I have similar problems with every scheme that embeds information about language-technical details (e.g., scope, storage class, syntactic category) into names. I agree that in some cases, building type hints into variable names can be helpful, but in general, and especially as software evolves, this becomes a maintenance hazard and a serious detriment to good code. Avoid it as the plague.

So, I don't like naming a variable after its type; what do I like and recommend? Name a variable (function, type, whatever) based on what it is or does. Choose meaningful name; that is, choose names that will help people understand your program. Even *you* will have problems understanding what your program is supposed to do if you litter it with variables with easy-to-type names like x1, x2, s3, and p7. Abbreviations and acronyms can confuse people, so use them sparingly. Acronyms should be used sparingly. Consider, mtbf, TLA, myw, RTFM, and NBV. They are obvious, but wait a few months and even I will have forgotten at least one.

Short names, such as x and i, are meaningful when used conventionally; that is, x should be a local variable or a parameter and i should be a loop index.

Don't use overly long names; they are hard to type, make lines so long that they don't fit on a screen, and are hard to read quickly. These are probably ok:

```
partial_sum    element_count    staple_partition
```

These are probably too long:

```
the_number_of_elements    remaining_free_slots_in_symbol_table
```

I prefer to use underscores to separate words in an identifier (e.g, element_count) rather than alternatives, such as elementCount and ElementCount. Never use names with all capital letter (e.g., BEGIN_TRANSACTION) because that's conventionally reserved for macros. Even if you don't use macros, someone might have littered your header files with them. Use an initial capital letter

for types (e.g., Square and Graph). The C++ language and standard library don't use capital letters, so it's int rather than Int and string rather than String. That way, you can recognize the standard types.

Avoid names that are easy to mistype, misread, or confuse. For example

```
name    names    nameS
foo     f00
fl      f1       fI       fi
```

The characters 0, o, O, 1, l, and I are particularly prone to cause trouble.

Often, your choice of naming conventions is limited by local style rules. Remember that a maintaining a consistent style is often more important than doing every little detail in the way you think best.

Should I put "const" before or after the type?

I put it before, but that's a matter of taste. "const T" and "T const" were - and are - (both) allowed and equivalent. For example:

```
const int a = 1;        // ok
int const b = 2;        // also ok
```

My guess is that using the first version will confuse fewer programmers (``is more idiomatic'').

Why? When I invented "const" (initially named "readonly" and had a corresponding "writeonly"), I allowed it to go before or after the type because I could do so without ambiguity. Pre-standard C and C++ imposed few (if any) ordering rules on specifiers.

I don't remember any deep thoughts or involved discussions about the order at the time. A few of the early users - notably me - simply liked the look of

```
const int c = 10;
```

better than

```
int const c = 10;
```

at the time.

I may have been influenced by the fact that my earliest examples were written using "readonly" and

```
readonly int c = 10;
```

does read better than

```
int readonly c = 10;
```

The earliest (C or C++) code using "const" appears to have been created (by me) by a global substitution of "const" for "readonly".

I remember discussing syntax alternatives with several people - incl. Dennis Ritchie - but I don't remember which languages I looked at then.

Note that in const pointers, "const" always comes after the "*". For example:

```
int *const p1 = q;      // constant pointer to int variable
int const* p2 = q;      // pointer to constant int
const int* p3 = q;      // pointer to constant int
```

What good is static_cast?

Casts are generally best avoided. With the exception of dynamic_cast, their use implies the

possibility of a type error or the truncation of a numeric value. Even an innocent-looking cast can become a serious problem if, during development or maintenance, one of the types involved is changed. For example, what does this mean?:

```
x = (T)y;
```

We don't know. It depends on the type T and the types of x and y. T could be the name of a class, a typedef, or maybe a template parameter. Maybe x and y are scalar variables and (T) represents a value conversion. Maybe x is of a class derived from y's class and (T) is a downcast. Maybe x and y are unrelated pointer types. Because the C-style cast (T) can be used to express many logically different operations, the compiler has only the barest chance to catch misuses. For the same reason, a programmer may not know exactly what a cast does. This is sometimes considered an advantage by novice programmers and is a source of subtle errors when the novice guessed wrong.

The "new-style casts" were introduced to give programmers a chance to state their intentions more clearly and for the compiler to catch more errors. For example:

```
int a = 7;
double* p1 = (double*) &a;                      // ok (but a is not a double)
double* p2 = static_cast<double*>(&a);  // error
double* p2 = reinterpret_cast<double*>(&a);     // ok: I really mean it

const int c = 7;
int* q1 = &c;                   // error
int* q2 = (int*)&c;             // ok (but *q2=2; is still invalid code and may fail)
int* q3 = static_cast<int*>(&c);        // error: static_cast doesn't cast away const
int* q4 = const_cast<int*>(&c); // I really mean it
```

The idea is that conversions allowed by static_cast are somewhat less likely to lead to errors than those that require reinterpret_cast. In principle, it is possible to use the result of a static_cast without casting it back to its original type, whereas you should always cast the result of a reinterpret_cast back to its original type before using it to ensure portability.

A secondary reason for introducing the new-style cast was that C-style casts are very hard to spot in a program. For example, you can't conveniently search for casts using an ordinary editor or word processor. This near-invisibility of C-style casts is especially unfortunate because they are so potentially damaging. An ugly operation should have an ugly syntactic form. That observation was part of the reason for choosing the syntax for the new-style casts. A further reason was for the new-style casts to match the template notation, so that programmers can write their own casts, especially run-time checked casts.

Maybe, because static_cast is so ugly and so relatively hard to type, you're more likely to think twice before using one? That would be good, because casts really are mostly avoidable in modern C++.

So, what's wrong with using macros?

Macros do not obey the C++ scope and type rules. This is often the cause of subtle and not-so-subtle problems. Consequently, C++ provides alternatives that fit better with the rest of C++, such as inline functions, templates, and namespaces.

Consider:

```
#include "someheader.h"

struct S {
        int alpha;
        int beta;
};
```

If someone (unwisely) has written a macro called "alpha" or a macro called "beta" this may not compile or (worse) compile into something unexpected. For example, "someheader.h" may contain:

```
#define alpha 'a'
```

```
#define beta b[2]
```

Conventions such as having macros (and only macros) in ALLCAPS helps, but there is no language-level protection against macros. For example, the fact that the member names were in the scope of the struct didn't help: Macros operate on a program as a stream of characters before the compiler proper sees it. This, incidentally, is a major reason why C and C++ program development environments and tools have been unsophisticated: the human and the compiler see different things.

Unfortunately, you cannot assume that other programmers consistently avoid what you consider "really stupid". For example, someone recently reported to me that they had encountered a macro containing a goto. I have seen that also and heard arguments that might - in a weak moment - appear to make sense. For example:

```
#define prefix get_ready(); int ret__
#define Return(i) ret__=i; do_something(); goto exit
#define suffix exit: cleanup(); return ret__

int f()
{
        prefix;
        // ...
        Return(10);
        // ...
        Return(x++);
        //...
        suffix;
}
```

Imagine being presented with that as a maintenance programmer; "hiding" the macros in a header - as is not uncommon - makes this kind of "magic" harder to spot.

One of the most common subtle problems is that a function-style macro doesn't obey the rules of function argument passing. For example:

```
#define square(x) (x*x)

void f(double d, int i)
{
        square(d);      // fine
        square(i++);    // ouch: means (i++*i++)
        square(d+1);    // ouch: means (d+1*d+1); that is, (d+d+1)
        // ...
}
```

The "d+1" problem is solved by adding parentheses in the "call" or in the macro definition:

```
#define square(x) ((x)*(x))     /* better */
```

However, the problem with the (presumably unintended) double evaluation of i++ remains.

And yes, I do know that there are things known as macros that doesn't suffer the problems of C/C++ preprocessor macros. However, I have no ambitions for improving C++ macros. Instead, I recommend the use of facilities from the C++ language proper, such as inline functions, templates, constructors (for initialization), destructors (for cleanup), exceptions (for exiting contexts), etc.

How do you pronounce "cout"?

"cout" is pronounced "see-out". The "c" stands for "character" because iostreams map values to and from byte (char) representations.

How do you pronounce "char"?

"char" is usually pronounced "tchar", not "kar". This may seem illogical because "character" is

pronounced "ka-rak-ter", but nobody ever accused English pronunciation (not "pronounciation" :-) and spelling of being logical.

Morgan Stanley | Columbia University | Churchill College, Cambridge

home | C++ | FAQ | technical FAQ | C++11 FAQ | publications | WG21 papers | TC++PL | Tour++ | Programming | D&E | bio | interviews | videos | applications | glossary | compilers