Speaking C++ As A Native†

Bjarne Stroustrup

AT&T Labs – Research

Abstract. C++ supports several styles ("multiple paradigms") of programming. This allows great flexibility, notational convenience, maintainability, and close-to-optimal performance. Programmers who don't know the basic native C++ styles and techniques "speak" C++ with a thick accent, limiting themselves to relatively restrictive pidgin dialects. Here, I present language features such as classes, class hierarchies,* abstract classes, and templates, together with the fundamental programming styles they support. In particular, I show how to provide generic algorithms, function objects, access objects, and delayed evaluation as needed to build and use flexible and efficient libraries. The aim is to give an idea of what's possible to provide, and some understanding of the fundamental techniques of modern C++ libraries.

INTRODUCTION

What can I tell you about C++ in physics computation? Clearly I can't tell you about physics; I'm not a physicist. Clearly I can't tell you about math, because I haven't practiced it since I got my degree. And clearly I can't tell you about all of the wonderful C++ programs and libraries that you have, because you know them and I don't.

Instead, I'm going to show a lot of little snippets of code and explain why I think they are interesting and useful. Before I do that, I'm going to talk a little about standard C++, assuming that you are not all up on the latest state of the C++ world.

WHAT IS C++?

C++ is a general purpose programming language. It can do more things than any language I've heard deemed "general purpose." It has a bias toward systems programming: you can hack device drivers with C++; it has even been used to program diesel engine fuel injectors. Perhaps most significantly, C++ is a multiparadigm programming language because it's meant to support a variety of ways of expressing yourself.

C++ supports:

- *C-style programming* — C++ is a better C, maintaining C's flexibility and run-time efficiency while improving type checking;
- *data abstraction* — the ability to create types that suit your needs;
- *object-oriented programming* — the idea of programming with class hierarchies and runtime polymorphism; and
- *generic programming* — programming using type parameterization on both data types and algorithms.

Why does C++ support these diverse approaches? Because the most effective styles of programming involve a variety of techniques that people often classify as belonging to different paradigms.

We've had an ISO standard (2) for about 2 years now. While a set of minor clarifications is supposed to be voted in next week,[1] we've been working hard on stability (rather than on changes or new extensions).

This has led to a lot of implementations, and they are converging to the standard. Thus, our ability to write code which is portable across operating systems and machine architectures is improving. Some of these implementations are even free; this is important to grad students and others who like to try new things. C++ works on almost all platforms, including all of the major ones.

† This paper is an abridged and edited transcript, prepared by Mark Fischler, Walter Brown, and Bjarne Stroustrup, based on the video recording of the plenary talk.

∗ For lack of space, the discussion of class hierarchy design and the use of abstract classes is missing from this transcript.

[1] The vote has since been held as scheduled: the resolutions approving a Technical Corrigendum passed unanimously. Currently in the hands of the Project Editor, the resulting ISO document will be formally issued shortly (we hope) after he is finished with it.

As a result, there are lots of foundation libraries, lots of scientific libraries, and lots of support for applications of various sorts and for lots of environments. However, here I'm going to show small elegant examples — the building blocks for the programming styles — because you can find just about anything else in a lot of other places.

USING A PROGRAMMING LANGUAGE

Ideals:

- *directness* — represent concepts directly in a program; and
- *independence* — represent independent concepts independently in a program.

If you have some ideas, you want to write them down so that your thoughts are reflected directly in the code. What you want to say, you want to say clearly. If your thoughts are muddled, you are going to get a lousy program. That's a different issue, and there the program language designer can help only marginally: by supporting clear thinking better than woolly, muddled thinking. That's very hard to do without becoming paternalistic and restrictive. C++ invariably errs to the side of allowing you to say more rather than on the side of allowing you to say just what I might consider good.

It should also be possible to represent independent ideas independently. The alternative is a big glob of code that does "everything" for you, but you can't figure out which part is connected to what. To avoid such messes, you try to keep separate concerns and ideas separate, so that if one thing needs to be changed, you can do so without changing lots of apparently unrelated things.

The *class* is the main construct in C++. It is used to express concepts. The *class* plays a lot of roles because there are lots of different kinds of concepts. We can have, for example, value types; function types; constraint declarations; resource handles; node types; interfaces; and many more.

A VALUE TYPE CLASS

One of the simplest examples I've come up with to illustrate some of these ideas is a simple value type, *Range*. A *Range* object holds a value guaranteed to be within specified bounds:

```
void f(Range& r, int n)
{
  try {
    Range v1(0,3,10);
    Range v2(7,9,100);
    v1 = 7;  // ok: 7 is in [0,10)
    int i = v2;  // extract value
    r = 7;   // may throw exception
    v2 = n;  // may throw exception
    v2 = 3;  // will throw exception:
             // 3 is not in [7,100)
  }
  catch(Range_error) {
    cerr << "Range error in f();"
  }
}
```

A value within the bounds is fine. However, you will not succeed if you attempt to enter a value that is not within bounds into a *Range*. In that case, you get an exception. That is, *Range* throws an exception. You can catch such exceptions, as shown above, and possibly recover from the error. That way, you can pass around values guaranteed to be within the specified bounds.

A *Range* is a very simple concept and I can express it quite directly in C++:

```
class Range { // simple value type
  int  value, low, high;

  // invariant: low <= value < high
  void check(int v)
  {  if (v<low || high<=v)
       throw Range_error();
  }
public:
  Range(int lw, int v, int hi)
    : low(lw), value(v), high(hi)
    { check(v); }

  Range(const Range& a)
  {  low=a.low;
     value=a.value;
     high=a.high;
  }

  Range& operator=(const Range& a)
  {  check(a.value); value=a.value; }

  Range& operator=(int a)
  {  check(a); value=a; }

  // extract value:
  operator int() const
  {  return value; }
};
```

This example embodies the very simplest idea of an object whose value is constrained. Notice the notion of the invariant established in a constructor to make sure that every *Range* object is valid and maintained by every member function. The function *check()* says that, if the given value is not in bounds, we throw an exception. Each function that sets a value that could be out of bounds *check()*s it first.

For example, the constructor uses *check()* when we make an object from a triple (*low, value, high*): If the initial value wasn't in bounds, the object will never be created — the constructor will fail. That's fine because then you don't have an invalid object to get yourself in trouble with later. Assignments, too, *check()* when needed.

The representation of a *Range* (i.e., the integers *value*, *low*, *high*) is *private* and only accessible by the functions declared in class *Range*. Note how construction (initialization) and assignment is specified by the programmer.

GENERALIZING A VALUE TYPE

I said that C++ is there to express ideas directly, but I didn't do quite what I said. While I'd said I was representing *thing*s in a range — and if the *thing* was not in the range you threw an exception — the code used *int*s, not "*thing*s."

The code can actually work for any type of *thing*, provided you can check the invariant that some *thing* is higher than *low* and less than *high*. And so I can generalize by saying that a *Range* is a range over values of type *T*, where *T* is anything that you can meaningfully check a range of. So I rewrote *Range*, not in terms of integers, but in terms of the arbitrary type *T*. This illustrates the C++ *template* concept:[2]

```
// simple value type
template<class T> class Range {
  T  value, low, high;
  // invariant: low <= value < high

  void check(const T& v)
  {  if (v<low || high<=v)
       throw Range_error(); }
public:
  Range(const T& lw,
        const T& v, const T& hi)
    : low(lw), value(v), high(hi)
    {  check(v); }

  Range(const Range& a)
  {  low=a.low;
     value=a.value;
     high=a.high;
  }

  Range& operator=(const Range& a)
  {  check(a.value); value=a.value; }

  Range& operator=(const T& a)
  {  check(a); value=a; }

  // extract value:
  operator T() const { return value; }
};
```

Now we can say that we want a range of integers, or a range of doubles, a range of characters, or even a range of *string*s:

```
Range<int> ri(10, 10, 1000);
Range<double> rd(0, 3.14, 1000);
Range<char> rc('a', 'a', 'z');
Range<string> rs("Algorithm",
  "Function", "Zero");
```

The *string* is the standard library string type. Of course you can compare *string*s: *string* comparison gives lexicographical ordering. It works. For example, here "Function" is between "Algorithm" and "Zero." So this generalizes nicely.

CONSTRAINTS

If you look back at the previous example, I still did not do exactly what I said. I'd said I was going to check ranges for any type *T for which it was meaningful to do comparisons*. But I didn't write that; I defined *Range* for an arbitrary type *T*. The construct *template<class T>* is the good old mathematical "for all *T*." How can we impose the constraint that our objects should have a linear ordering?

You can rely on the compiler to check. Code like this will not compile if you feed it a type for which < or <= doesn't work. However, the error message can be very verbose and cryptic, so let's try to express this constraint directly.

I want to ensure, for the class *Range<T>*, that *T* is comparable and that *T* is assignable. How — in Standard C++ — can I express that? Here is one way:

```
template<class T> struct Comparable {

  // the constraints check:
  static void constraints(T a, T b)
```

[2] Templates are often considered new because I didn't invent them until 1988, but the world can be slow to catch on to new ideas.

```
    {  a<b; a<=b; }

    // trigger the constraints check:
    Comparable()
    {  void (*p)(T,T) = constraints; }
};

Template<class T> struct Assignable {
    // ...
};

template<class T> class Range
    : private Comparable<T>,
      private Assignable<T> {
    // ...
};

Range<int> r1(1,5,10);  // ok
Range< complex<double> > r2(1,5,10);
    // constraint error: no < or <=
```

I define a little template class *Comparable*, which will compile if *T* fulfills the criteria I defined, namely that if you have two *T*'s, you should be able to compare them using < and <=. The *constraints()* function just checks that constraint. It doesn't do any real work; it just expresses the constraint by exercising the aspect of the type that I am interested in. The constructor makes sure *constraints()* is exercised: it can make a *Comparable<T>* if and only if *constraints()* can be compiled for the type *T*. As you will see, the compiler never actually generates any code for this. I write *Assignable* in the same way.

Notice there are no macros and no magic here. Furthermore, it is pretty minimal:

- I have a single line that names the property I want to check;
- I have a single line that expresses that check; and
- I have a single line that expresses when it's checked.

This is not particularly new; I wrote about constraints in *The Design and Evolution of C++* (6) in 1994, but this is the first time I have been able to express general constraints in small and simple code snippets like this.

Now, when I take a range of *thing*s, the compiler checks whether *thing*s are comparable. Compilers can compile this so that it's all a compile-time effort with no run-time effect. So when I talk about representing concepts as classes, it doesn't mean that I have to create objects in the machine representing the concepts and invoke operations on them to get work done. It simply means that I can express my concept and have it work.

Of course *int*s are comparable, so *Range<int>* compiles. Next, we try for a range of double-precision complex numbers, but you can't make a *Range< complex<double> >* because we check *Comparable< complex<double> >*: we try to do a <, but that doesn't work — operator < is not defined for complex numbers — so we get a compile-time error.

A compiler will check anything you do to a template parameter class *T* even if you don't write specific constraints checks. However, if you've ever tried, you know that the error messages leave a lot to be desired. The main point of the *constraints()* technique is to make the constraints explicit. Doing that yields good, specific error messages and, importantly, it allows us to express a very general notion of constraint.

Anything you can say in the language you can check in a constraint. In particular, it is easy to express constraints involving more than one type. For example, if you have a template with three type arguments, *T1*, *T2*, and *T3*, you can say that the result of multiplying a *T1* by a *T2* should be assignable to a *T3* by simply saying *t3=t1*t2* for suitably declared *t1*, *t2*, *t3*. Because you express this in the language itself, rather than in some language designed to express constraints, this technique is actually more powerful than what is found in non-research languages and in most research languages. And still, expressing a constraint is four simple lines of code; no magic is required.

MANAGING RESOURCES

One thing that comes up again and again in my world is that there are a lot of resources to take care of. Memory is the one resource people always talk about, but more critical are things like file handles, thread handles, and sockets. It doesn't actually matter if you can clean up all of your memory if you have left thread handles hanging around, because they own memory you can't clean up.

In general, it's very, very difficult to deal with resources. In practice, however, most resources live in a scope and this is the simple and common case that I'm going to show a solution for.

The general structure of the solution is to acquire a resource by initializing some object that holds it. So we are introducing a class — a resource handle class — that represents the notion of the resource. The class controls access to the resource. Of course classes are good at that that kind of control. You can access the resource only by using functions that the handle provides for that — representations are private. Creation is controlled by constructors, copying can be controlled, and final cleanup is provided by destructors. A destructor is a function that is guaranteed to be invoked upon exit from the scope of a variable, and it just releases the resource at that point.

Actually, this technique is the key to exception safety, but I don't have time here to go into that in detail. If you are interested, read Appendix E of *The C++ Programming Language, Special Edition* (5).[3]

To illustrate, I sketch a piece of code I've seen many times in many versions in C and C++ programs: you grab something, you use it, and then you release the resource:

```
void my_fct(const char* p)
{
  FILE* f = fopen(p,"r"); // acquire
  // use f
  fclose(f);              // release
}
```

This is fine as long as you actually get to releasing the resource. If a C program does a *longjump* here you're in trouble; if a C++ code *throw*s an exception here you're in trouble. In short, this is very simple but very unsatisfactory code.

The naive fix that everybody uses when they first start playing with exceptions is to wrap a *try* block around the resource's initialization and use:

```
void my_fct2(const char* p)
{
  FILE* f = 0;
  try {
    f = fopen(p,"r");
    // use f
  }
  catch () {  // handle exception
     // ...
  }
  if (f) fclose(f);
}
```

This is fine, but I find that it is only fine if you apply the technique consistently and correctly. Here's what we just did:

- We found a problem in the code, a problem caused by people failing to think things through and take care of all the error conditions.
- Then we solved it by doubling the size of the code and complicating the control structure.

The chance of forgetting something and not getting things right is at least linear with the size of the code. So if I recommended this, I would be recommending a way of dealing with careless errors that doubled the probablity of careless errors. This particular problem actually held back exceptions in C++ for at least a year.

Remember: if I have a concept, I'm suppose to represent it directly by a class. So I create a little handle class to represent my notion of an open file:

```
// in some support library:
class File_handle {
  FILE* p;
public:
  File_handle(const char* pp,
              const char* r)
    :p(fopen(pp,r))
    { if (p==0) throw Bad_file(); }

  File_handle(const string s,
              const char* r)
    :p(fopen(s.cstr(),r))
    { if (p==0) throw Bad_file(); }

  ~File_handle()  // destructor
  {  if (p) fclose(p); }

  // access functions:
  // ...
};

void my_fct3(string s)
{
    File_handle f(s,"r");
    // use f
}
```

The constructor creates the handle and opens the file. If *open()* succeeds, all is fine. If *open()* fails, we don't create the handle and exit *my_fct3()* throwing an exception. The destructor releases the resource — here, it closes the file — if you managed to acquire it. The handle class is the kind of stuff you stick in a support library. However, if you have a resource that nobody else has, you have to write a resource manager yourself. That will be maybe ten lines of code that you write once and then use wherever you acquire one of those resources. You typically acquire a kind of resource in many places in your code, but you need to write only one class to handle that safely. On the other hand, if you use the *try*-block approach, you have to get the error handling code right in every case.

The way you now write your code is to create a handle for the file named *s* with read access. Then you use the file through that handle. You don't have to explicitly close the file because that's taken care of by the handle's destructor. So I've simplified the code while making it exception-safe. The chances of making mistakes are now much more limited.

[3] If your book doen't have an appendix E, just go to my home pages (9) and download a version, or augment Bjarne's retirement fund by buying a new copy :-).

CONTROL ABSTRACTION

There is a related problem that was open, I estimate, for about 20 years. It's trying to deal with the fact that a lot of the code we write is of the form "Do some prefix code, then do the real thing, then do some suffix code." Common examples of that include lock/unlock, transaction-start/transaction-commit, debug-trace-on/off, and "acquire the resource, do the operation, release the resource." If you've written a large program, you'll have code of this general style somewhere.

As shown below, I want to take some arbitrary class X — say one you are going to write tomorrow so I couldn't possibly know what it is — and wrap it so that prefix code and suffix code is implicitly done. When I write *x->count()*, it should translate into *prefix()*, *count()*, *suffix()*, and similarly for other member functions of your class. Further, I shouldn't have to know what *prefix()* and *suffix()* are when I write the wrapper class:

```
void f(X& x)
{
  Wrap<X> xx(x,prefix,suffix);
  int n = xx->count();
    // prefix(); n=x.count(); suffix();
  xx->g(99);
    // prefix(); x.g(99); suffix();
}
```

This is close to what people mean when they say "control abstractions." It does something to the control flow in your program, in a guaranteed and declarative manner.

The constraints on a solution to this problem are that there should be optimal performance: It should be possible to inline *prefix()* and *suffix()*. This is very important if these are, say, assembly code that does lock/unlock. It has to work for pre-existing X's. Oh, and by the way, I can write it in 16 lines of standard C++:[4]

```
template<class T, class Suf>
class Wrap_proxy {
  T* p;
  Suf suffix;
public:
  Wrap_proxy(T* pp, Suf s)
   :p(pp), suffix(s) {}

  ~Wrap_proxy() { suffix(); }

  T* operator->() { return p; }
};
```

```
template<class T, class Pre, class Suf>
class Wrap {
  T* p;
  Pre prefix;
  Suf suffix;
public:
  Wrap(T& x, Pre pref, Suf s)
    :p(&x), prefix(pref), suffix(s) {}

  Wrap_proxy<T,Suf> operator->()
  { prefix();
    return Wrap_proxy<T,Suf>(p,suffix);
  }
};
```

This wraps an object by storing away the prefix and the suffix and a pointer to the object. Whenever you call the resulting wrapper object using operator arrow, the prefix code is invoked, then a proxy is created and the proxy's *operator->()* is called. When you are finished with the proxy, it is of course destroyed, calling the suffix code. This works. I present it here to show that

- you can do some control abstraction in C++, and
- the range of notions you can represent as a class is much wider than most people are willing to believe.

I describe the wrapper further in a paper (8).

GENERIC PROGRAMMING

Now I'm going to explain some of the basics of generic programming as it is represented in the C++ standard library. The first idea is that you can make yourself a lot of useful containers, such as *vector<T>*, *list<T>*, and *map<K,V>*. Since the standard library has all these and more, and since these containers' quality is really quite good, you can just use them without having to write them yourself.

Further, there are some very common things — you can find them in Knuth or Sedgwick — that we frequently do to all kinds of containers:

- find an element in a container,
- sort a container,
- perform an operation on each element of a container,
- remove elements that meet a given criteria from a container,
- copy a container.

[4] The example looks longer because some of the lines are artificially wrapped to fit the two-column format.

You don't want to hand-code these algorithms each time you use them. That would just be a waste, a nuisance, and a well-known source of bugs. People don't write their own sorts anymore, except for the few people who actually have a chance of getting them right. Similarly with the other basic algorithms, so they're provided in the standard library.

And we don't want to repeat the code for each algorithm for each container. That would be a nuisance and a maintenance hazard.

The standard library is organized as a framework of containers and algorithms. This organization is the work of Alex Stepanov. The problem was "how do you provide an algorithm over a set of containers that includes the Standard ones as well as those you have defined yourself?" The key idea is to say that any container can be seen as a sequence of elements. A sequence has a beginning and an end, and if you have access to an element you can get to the next element. That's all there is to it.

```
begin                  end
  |                     |
  v                     v
elem -> elem -> ... -> 0
```

Something that refers to an element of a sequence is called an *iterator*. The obvious C-style notation is that ++ for an iterator means "refer to the next element" and * means "get the value of the element referred to." This sequence notion is very general: it covers vectors, it covers lists, it covers trees. While the implementation of these notions may be different, the semantics of getting to the next element and getting the value of the next element are independent of what kind of data structure you are talking about as long as you can view the elements as part of a sequence.

Here is some pseudo-code expressing what we want to do: copy a sequence from begining to its end onto output, find the value in the sequence, and count the number of occurrences of the value in the sequence; we want to make it into real code:

```
// copy sequence to output:
copy(begin,end,output)

// find value in sequence:
find(begin,end,value)

// count occurrences of value
// in sequence:
count(begin,end,value)
```

One of the ideals of programming is the idea of direct representation of ideas in code. Given this pseudocode, what is the smallest step we could do to turn it into real code? Well, that would be doing nothing. We can't quite do that in C++: we have to put a semicolon after each expression to make it a statement.

We are also getting close to the other major ideal here: to represent independent concepts independently in the code. Notice what we have kept independent here.

- *Container type*: When we look at elements, we don't have to know which kind of container we are looking at. There's the notion that a container should have a beginning and an end, and — given its notion of begin and end — we can start at the beginning and examine each element until we reach the end. That's all that's required of a container.

- *Element type*: Element types are independent of the container types. A type is not required to be part of some class hierarchy to be used for elements in a container. The container notion does not intrude on the notion of an element.

- *Algorithm*: We separate algorithms from the containers. An algorithm need not be a member of a (container) class.

- *Comparison criteria*: When we do anything interesting with algorithms, we have comparison criteria, polices, and such. Each can be independently specified.

We can vary these four things (containers, elements, algorithms, and policies) independently. This is what allows the standard library to be five or six thousand lines of code, yet to do more than many libraries 20 times its size.

Let me show you some code for a simple linear search to find, in the sequence from *first* to *last*, the value *val*:

```
template<class In, class T>
// find val in sequence [first,last):
In find(In first, In last, T val)
{
  while (first!=last && *first!=val)
      // while we haven't reached the end
      // and haven't found what we seek
    ++first;  // carry on
  return first;
}
```

This is real code: The standard library looks like this. We've paramenterized *find()* so that we don't need to know which kind of iterator is used to represent the sequence. The type of element is another parameter.

So, we go through a loop until we have reached the end or found what we are looking for. As long as we haven't reached the end and as long as we haven't found the value we are looking for, we make the iterator refer to the next element and try again.

You may or may not like C or C++ syntax, but this is colloquial. If you want to deal with this class of languages, you'd better get used to it. Familiarity is often confused with what is natural. I don't think I'm doing that: this notation is not natural, but it's familiar to a lot of people. People can come to love it; I'm not sure they should, but they do.

What we can do now is to take a vector of integers and, say, apply *find()* to it for some value *x*, Did we hit the end? If so, *x* wasn't there; otherwise, we found *x*:

```
void f(vector<int>& v, int x)
{
  vector<int>::iterator p
    = find(v.begin(), v.end(), x);
  if (p != v.end())
    { /* we found x */ }
  // ...
}
```

Since this is a vector, the iterator is almost certainly implemented as an ordinary pointer. So *++first* simply makes *first* point to the next element in the vector. It's a standard machine instruction that adds a constant to a pointer. That's simple and efficient. Looking for the value **first* means dereference a pointer. If you measure this code, you will find that it's optimal; you cannot write better-performing code in C.

Now, let's try with a list of strings. I try to find the string *s* in it, using *find()*:

```
void f(list<string>& lst, string s)
{
  list<string>::iterator q
    = find(lst.begin(), lst.end(), s);
  if (q != lst.end())
    { /* we found s */ }
  // ...
}
```

An iterator for a list is unlikely to be implemented as a pointer to an element. It's going to be a pointer to some kind of link node. When we do a comparison here, it compares two link nodes. That's fine; that's still a simple and efficient pointer comparison. When I dereference — when I want to get a value — I grab into that node to extract its value field. When I increment the list iterator to get to the next element, I indirect through a "next field" to the link node for the next element. Again, you can see that this is exactly the code you would have hand written in any language you care to use: C, assembler, C++, whatever.

When we get out of the loop, we've found *s* or we've reached the end. The *find()* algorithm is really basic. However, there's lots and lots of code like that in the C++ standard library. It's deceptively simple, but it is fast and it is general.

Looking for a specific value is a special case of looking for something that meets some criteria. In my work, I more often look for something that fulfills a predicate *P*. That is, I'm not looking for a specific value such as 7, I'm looking for a value less than a threshold, or higher than a threshold, or something like that. So, I want to specify a predicate, something that express my criteria:

```
template<class In, class Pred>
In find_if(In b, In e, Pred p)
{
  while(b!=e && !p(*b))
      // while we haven't reached the end
      // and haven't found what we seek
    ++b; // carry on
  return b;
}
```

We just replace the earlier "not equal to" by "not meeting my criteria," and all of the code works again. Of course the *find()* function is just a simplification of *find_if()* where *P* is "equals." Here, I look in a vector of strings *v* for a string *"foo"*, using a predicate *less than "foo"*:

```
void f(vector<string>& v)
{
  vector<string>::iterator p
    = find_if(v.begin(), v.end(),
              Less_than<string>("foo"));
  if (p != v.end())
    { /* found: *p < "foo" */ }
  // ...
}
```

We go through the vector, from the beginning to the end, looking for something that's *less than "foo"*. If we didn't reach the end, we found something that meets that criteria and p now points to an element that did. This generates what I would consider the obvious code.

We can use *find_if()* for a list of records, where we want to check that the name field in the record is equal to that of a record that I'm interested in. For a lot of data processing that is exactly what you need: you check a notion of equality which is not the equality of the value of the record, it's the equality of some field of the record. Here, some notion of name-equality is used:.

```
void f(list<record>& lst,
       const Record& my_rec)
{
```

```
  list<Record>::iterator q
    = find_if(lst.begin(), lst.end(),
              Name_eq(my_rec));
  if (q != lst.end()) {
    // found: *q has same key as my_rec
  }
  // ...
}
```

FUNCTION OBJECTS

I have illustrated a general form of flexibility. *Name_eq* is the archetype of a predicate: it holds a value that you compare against. That value is stored when you construct the *Name_eq* object, and *operator()* — the application operator — simply does the comparison:

```
class Name_eq {
  const string s;
public:
  Name_eq(const Record& r)
    : s(r.name) {}

  static bool
  operator() const (const Record& r)
  { return r.n == s; }
};
```

We use *Name_eq* like this:

```
void f(list<record>& lst,
       const Record& my_rec)
{
  // ...
  find_if(lst.begin(), lst.end(),
          Name_eq(my_rec));
  // ...
}
```

For each element in *lst*, the predicate objects created by *Name_eq(my_rec)* is invoked. That function object compares the name field of the current element with the copy of name field of *my_rec* that was stored away by *Name_eq(my_rec)*.

Here is an archetypal function object. Such an object has a state that is established when you construct that object and that is used (in the application operator, *operator()*) as you go along:

```
template<class S> class F {
  S s;  // state
public:
  F(const S& ss) : s(ss)
  { /* establish initial state */ }

  void operator() (const S& ss)
  { /* do something with ss to s */ }

  // reveal state:
  operator S() { return s; }
};
```

This is very general. It is more general than a function because a function cannot be initialized to work against a contained state: A function object, in contrast, can carry state and you can extract the state from it. By parameterizing algorithms with such function objects you can express arbitrary predicates and policies.

It's quite common to pass a function object along, updating its state by an operation on each element of a container. The simplest example of that is to take a sum: you initialize a sum object to zero in its constructor — this would be the state, here a numeric value. As you go along, you add elements of the container to that value. When you are finished, you extract the resulting value sum from the sum object.

Interestingly, function objects also run faster than equivalent functions because little function objects inline better than functions. The reason is that when you pass a function you are passing a pointer to function and optimizers are not very good at dealing with pointers. On the other hand, if you pass a function object, you're passing an object rather than a pointer; when you do the operation on the object, you have the object, you have the function, and inlining is easy.

This is the reason that the generic general *sort()* in C++ often runs several times faster than *qsort()* in C. I have measured it from 2 to 7 times faster on things like floating-point numbers and simple strings. It's not really magic, these generic programming techniques just fit better with compiler technology than do C-style parameterization with pointers to functions.

DELAYED EVALUATION

You can use function objects directly. However, sometimes you'd like to use "the natural notation." That often means using operators like +, -, and *. In particular, we often want to express vector and matrix manipulation using conventional notation, e.g. $v=m*v2+v$. In addition, we want to evaluate such experessions without using temporaries, and without having expensive function calls compromise your run-time performance. The point of the following example is partly to avoid temporary values and partly to show you how to get the "natural" notation without overhead.

```
Matrix m;
```

```
Vector v, v2, v3;
// ...
v = m * v2 + v3;
```

The basic implementation idea is to generate a single function, *mul_add_and_assign(v,m,v2,v3)*, that knows that it's supposed to multiply, add, and assign. If this is on some form of CRAY you can write very beautiful code vectorizing such compound operations, but given only *v=m*v2+v3*, compilers are generally not smart enough to vectorize without help from the programmer. To help, we write something like this:

```
struct MV { // object representing
            // the need to multiply
  Matrix* m;
  Vector* v;
  MV(Matrix& mm, Vector& vv)
    : m(&mm), v(&vv) {}
};

MV operator * (const Matrix& m,
               const Vector& v)
{ return MV(m,v); }

MVV operator + (const MV& mv,
                const Vector& v)
{ return MVV(mv.m,mv.v,v); }

v = m*v2+v3;
  // v = MVV(MV(m,v2),v3);
  // mul_add_and_assign(m,v2,v3,v);
```

We make a little function object *MV* that simply keeps track that it has seen an *m* and a *v*. This represents the notion that *m* wants to be multiplied by *v*. We have operator *, given a matrix and a vector, make one of these *MV* objects. *MV(m,v)* expresses the notion that *m* would like to be multiplied by *v*. We do a similar thing for operator +: if you get an *MV* and a vector, it creates an *MVV* object that holds the matrix and the two vectors. So when we execute *m*v2+v3*, we just construct little objects until we have *MVV(MV(m,v2),v3)* — unravelling the expression collecting information — and in the end we use the collected information to generate *mul_add_and_assign(v,m,v2,v3)*.

The above example collects references to matrices and vectors. I chose matrices and vectors for this example because I know that lots of people must use large vectors and matrices. Copying a 10000x10000 matrix is expensive to most people. Another example of the delayed evaluation technique is to collect the value, the format, and an output stream so that when all of these things are together I can output the value with the right format onto the stream. Again, this relies on function objects. The whole thing is done by creating little function objects to hold the information until you got to the final function. Inlining is very important. So is pass-by-value, because these function objects get passed along, then the optimizer gets them, and they just disappear.

Function objects tend to be templates. An example here would be a matrix of *double*s stored densely: *Matrix<double,Dense>*. Most current C++ vector and matrix libraries work with these techniques, so they generate fast code. This is why many of you have seen graphs comparing the performance of Fortran and C++, with C++ winning. If you haven't seen such graphs, look for links on my C++ page. That was "known" to be impossible, but it's always nice to disprove a myth. These libraries' vector and matrix classes have little "policy objects" associated with them so a matrix is not just a matrix of *double*s, it's also something that controls, for further optimization, the way elements are stored and accessed. These "policy objects" controls need only be seen by expert users who care.

CLASS HIERARCHIES

I've spoken about C++ at length, giving a variety of examples, yet I haven't shown a single class hierarchy. According to some people's definition of OO, this means I haven't yet talked about Object-Oriented Programming. I should do so, because OOP is important and because some of the most interesting and important uses of C++ are in application domains that use class hierarchies effectively. However, object-oriented programming is the use of C++ that people know best — at least they think they know best. So here I have emphasized the other programming styles.

I think one of the keys to modern C++ is lots of little objects, as opposed to huge hierarchies. One of the reasons that hierarchies get large and massive is that you throw too much into them. Little objects representing policies, values, constraints, etc., are very useful and can provide generality, flexibility, and efficiency. In particular, "little objects" can be used to design leaner hierarchies by not relying exclusively on facilities represented within a hierarchy.

I do not want to be misunderstood: class hierarchies and their use in object-oriented programming are important. Lack of space, unfortunately, keeps me from describing this last piece of my talk in this transcript (but it is in the video of my talk that you can view from the ACAT2000 conference website).

SUMMARY

Try to think of C++ as a new language. A lot of you have used it for a long time; you will know that there are techniques that didn't work a few years ago when you last tried. A lot of these now work.

This is a good time to be adventurous because the standard is out, the compilers are starting to support the standard, not just in language features but also in terms of efficiency. On the other hand, of course, be careful! Not every technique works for every project and for every group of people. But this is a good time to start to see what concepts you can express more directly and more efficiently than before.

For those of you who are beginning with C++, please remember that C++ is not just C with a few useless and inefficient bits added; you can write cleaner, shorter, and faster-running code in C++ than in C if you know how. An example is sorting: the general, generic, and type-safe *sort()* in the C++ standard library is not just easier to use than the C-style *qsort()*, it is often several times faster. And C++ is not just class hierarchies, there is a lot more to it. A lot of modern C++ techniques are focussed on templates, containers, and function objects.

Prefer the C++ standard library style to traditional C style; it is simply easier to express ideas using *vector*, *list*, and *string*, rather than with arrays, pointers, and casts. If you are not careful, you can get overhead in both cases. You have to understand things to write good code; you can't just blindly plow along in either the C++ style or the C style and expect to produce efficient and maintainable code.

FOR MORE INFORMATION

There's a lot of reference material available. You can look at my "Third Edition" book (5) — the "Special Edition" is the hardcover version — or you can get the Standard itself (2) via the web. My *Special Edition* was updated last year from my *Third Edition*: I added another 100 pages, corrected many errors, and clarified numerous issues. I'm now confident enough to offer $16 for every new bug reported to me. I haven't yet been ruined.

The Design and Evolution of C++ (6) is for people who are interested in why things are the way they are. Answers to many "why?" questions about the design of C++ can be found there. It is the closest thing we have for a rationale for the design of C++.

I'm the editor of Addison Wesley's *C++ In Depth* series. I'll mention two of the books here. One is Herb Sutter's book on exceptions (10), which gives a lot of exercises and discussions, going into greater detail about exception handling techniques than I do in my *Special Edition*. Another is Andy Koenig and Barbara Moo's book called *Accelerated C++* (3) which basically is a tutorial on modern C++; it is probably the first such introduction. It introduces templates four chapters before it introduces pointers. This gives you an idea of how much the world has changed.

There are some papers on the web. In (7), I do a micro-analysis of some very simple C and C++ examples used in education. The results were good enough — from a C++ perspective — to cause a firestorm of letters to the editor when it was published last year. I consider it non-controversial. The code is available on my web site (9) so you can run it yourself.

There are many useful links on my C++ page (www.research.att.com/ bs/C++.html). In particular, the ACCU site (1) has many useful book reports. These reviews are done by professionals and are reasonably unbiased — as opposed to most reviews that you find on the web. Many of my favorite links can be found on my home pages: FAQ's, the standard itself, compilers, garbage collectors, papers, book chapters, etc.

REFERENCES

1. Association of C and C++ Users. `www.accu.org`
2. International Standard Organization, *The C++ Programming Language*, 1998.
3. Koenig, Andrew and Barbara Moo, *Accelerated C++*, Addison Wesley Longman, 2000.
4. Stroustrup, Bjarne, *Learning Standard C++ as a New Language*, C/C++ Users Journal. pp 43-54. May 1999.
5. Stroustrup, Bjarne, *The C++ Programming Language, Special Edition*, Addison Wesley Longman, 2000.
6. Stroustrup, Bjarne, *The Design and Evolution of C++*, Addison Wesley Longman, 1994.
7. Stroustrup, Bjarne, *Why C++ isn't Just an Object-oriented Programming Language*, Addendum to OOPSLA'95 Proceedings. OOPS Messenger. October 1995.
8. Stroustrup, Bjarne, *Wrapping C++ Member Function Calls*, The C++ Report. June 2000, Vol 12/No 6.
9. Stroustrup, Bjarne, `www.research.att.com/~bs`
10. Sutter, Herb. *Exceptional C++*, Addison Wesley Longman, 2000.