Foundations for Native C++ Styles

Andrew Koenig
Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Over the past decade, C++ has become the most commonly used language for introducing object-oriented programming and other abstraction techniques into production software. During this period, C++ has evolved to meet the challenges of production systems. In this, C++ differs radically from languages that come primarily from academic or research environments, and from less widely used languages. Although C++ has also been extensively used in academia and for research, its evolution was driven primarily by feedback from its use in industrial applications.

In this paper, we focus on three design areas key to successful C++ use. In doing so, we explore fundamental C++ concepts and facilities and present distinctive C++ design and programming styles that have evolved to cope with the stringent demands of everyday systems building. First we explore C++'s support for concrete data types and containers and give examples of how the C++ generic programming facilities, together with well-designed libraries, can yield flexibility and economy of expression. Next we examine some uses of class hierarchies, touching on issues including encapsulation, interface design, efficiency, and maintainability. Finally, we note that languages succeed for reasons that are not entirely technical and review the background for C++'s success.

This paper is not a C++ tutorial. However, it does include enough code examples and supporting commentary that readers familiar with programming languages in general but unfamiliar with C++ can grasp the key C++ language constructs and programming techniques.

1 Introduction

C++ was designed to combine the strengths of C as a systems programming language with Simula's facilities for organizing programs. During the 60's and 70's, the key concepts, techniques, and language features for what came to be known as ''object-oriented programming'' and ''object-oriented design'' had developed in connection with the Simula language. During the 80's, C's close-to-the-machine semantics gave it the edge in run-time and space efficiency, portability, and flexibility that established C as the dominant systems programming language.

Thus C++ started from a sound theoretical and practical basis. Feedback from widespread use guided its further evolution. C++ supports the design and efficient implementation of elegant programs from toy examples to very large systems.

Over the years, distinct C++ styles of design and programming have evolved. This evolution has progressed to the point where we can identify and explore key notions and techniques.

2 Extending C's Model of Systems Programming

A fundamental property of computers in widespread use has remained remarkably constant: memory is a sequence of words or bytes, indexed by integers called addresses. Modern machines—say, designed during the last 20 years—have in addition tended to support directly the notion of a function call stack. Furthermore, all popular machines have some important facilities, such as input-output, that do not fit well into the conventional byte- or word-oriented model of memory or computation. These facilities may require special machine instructions or access to ''memory'' locations with peculiar addresses. Either way, from a higher-level language point of view the use of these facilities is ''messy'' and machine-architecture-specific.

C is by far the most successful language designed to exploit such computers by providing the programmer with a programming model that closely matches the machine model. C directly provides language-level and machine-architecture-independent notions that directly map to the key hardware notions: characters for using bytes, integers for using words, pointers to use the addressing mechanisms, functions for program abstraction, and an absence of constraining language features so that the programmer can manipulate the inevitable messy hardware-specific details. The net effect has been that C is relatively easy to learn and use in areas where some knowledge of the real machine is a must or simply a benefit. Moreover, C is easy enough to implement that it has become available virtually everywhere.

The other main trend in programming languages—particularly in the academic community—has been to try to define a machine-independent semantics for a language. The goal is to get the language away from the messy details of computer hardware such as bytes and pointers, and allow programmers to operate in a provably sound and logically simple universe. Sometimes, this is expressed as providing an ideal virtual machine for the programmer. Lisp is the most prominent of these languages, but almost all high-level language designs aim to hide the fundamentals of the underlying machine from the programmer.

The results from this idealistic school of language design have not been uniformly encouraging. The semantic bases of such languages are cleaner and simpler than that of languages such as C. However, each language family has a different semantic base, which means that a systems programmer in one of these languages must learn both the high-level semantic base of the programming language and the low-level one of the machine. Consequently, even if learning the higher-level language is easier than learning C, learning C has for many proven easier than learning the higher-level language plus the machine model. This trend has been amplified where operating systems add their own layer of interfaces and conventions to the set of concepts to be mastered. Traditionally, these interfaces and conventions are close to the machine in the way C is. Indeed, operating systems sometimes describe their interfaces only in C terms, leaving the machine model unspecified beyond that.

Generating code for a language with semantics that are not close to the traditional bytes-and-pointers machine architectures has been a constant problem. It is not unusual to pay a factor of three or ten in runtime for a higher-level semantics, especially when those semantics include the notion that the type of an expression is not knowable until that expression is evaluated during program execution.

If C gains its advantages from a match with traditional machine architectures, perhaps it is reasonable to try to design new machine architectures to fit higher-level languages. As a result, there has been a steady stream of high-level machine architectures for particular families of languages. Unfortunately, this approach has repeatedly run aground on two rocks. Each higher-level language family has a different semantic base so that a machine optimized for one such language base becomes useless—or at least uneconomical—for all other such languages as well as for traditional languages such as C and Fortran. Also, as the semantic base of a higher-level language is extended to deal with the various forms of messiness necessary to deal with a complete computer system (e.g. multiple users, I/O devices, security, hardware diagnostics) some of that messiness leaks into the semantic base itself—thereby diluting its appeal.

Worse yet, almost everybody who pays for computers uses traditional languages and all large hardware manufacturers have a strong financial interest in traditional architectures. Therefore, the best tools, the largest number of hardware designers, the newest hardware technologies, the best production facilities, and the most money is spent on bytes-and-pointers architectures. Consequently, high-level machine architectures tend to be a generation or two behind the state of the art and never become cost effective.

C++ was designed to dodge the dilemma that machine-level language semantics, as in C, had a fundamental advantage, yet the programming model offered by languages such as C constrained the kinds of applications that could be successfully built. The solution chosen for C++ was to augment the low-level

language features with powerful, yet affordable abstraction mechanisms [Stroustrup,1985]:

> ''A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed and a set of concepts for the programmer to use when thinking about what can be done. The first aspect ideally requires a language that is ''close to the machine,'' so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second aspect ideally requires a language that is ''close to the problem to be solved'' so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.''

Given both machine-level facilities and abstraction mechanisms, there is a danger of opening a semantic gap between the two sets of facilities. That is, a language might offer the programmer the choice between writing efficient code (using machine-level facilities) or elegant code (using abstraction). This was not considered an acceptable choice to offer C++ programmers. It was essential to provide abstraction facilities that could be used to write user-defined types with little or no overhead compared to C or even assembly code. To be useful in this context a mechanism can't just be elegant, it must also be affordable.

3 Efficient User-defined Types

Small heavily used abstractions are common in many applications. Examples are characters, integers, floating point numbers, complex numbers, points, pointers, coordinates, transforms, (*pointer,offset*) pairs, dates, times, ranges, links, associations, nodes, (*value,unit*) pairs, disc locations, source code locations, BCD characters, currencies, lines, rectangles, scaled fixed point numbers, numbers with fractions, character strings, vectors, and arrays. Every application uses several of these; a few use them heavily. A typical application uses a few directly and many more indirectly from libraries.

C and other programming languages directly support a few of these abstractions. However most are not, and cannot be, supported directly because there are too many of them. Furthermore, the designer of a general-purpose programming language cannot foresee the detailed needs of every application. Consequently, mechanisms must be provided for the user to define such small concrete types. It was an explicit aim of C++ to support the definition and efficient use of such user-defined data types very well. They were seen as the foundation of elegant programming. As usual, the simple and mundane is statistically far more significant than the complicated and sophisticated.

Here is a declaration of a `Date` class:

```
class Date {
public:        // public interface:

        enum Month { jan, feb, mar, apr, may, jun,
                     jul, aug, sep, oct, nov, dec };

        Date(int d, Month m, int y);  // constructor

        // functions for examining the Date:

        int day() const;
        Month month() const;
        int year() const;

        string string_rep() const;     // return string representation
        void char_rep(char s[]) const; // place C-style string represen-
                                       // tation in C-style array s
```

```
        // functions for changing the Date:

        Date& add_year(int n);   // add n years
        Date& add_month(int n);  // add n months
        Date& add_day(int n);    // add n days
private:
        int d, m, y;  // representation
};
```

C++ uses { } to delimit scopes and // to start comments. ''Class'' is the C++ term for a user-defined type. In declarations, a suffix ( ) is used to specify a function, and a suffix & means ''reference to.'' Thus,

```
Date& add_year(int n);
```

declares a function taking an integer argument and returning a reference to a `Date`. The type `void` is used to specify that a function doesn't return a value.

The set of operations is fairly typical for a user-defined type:

[1] A constructor specifying how objects/variables of the type are to be initialized. In this case, a `Date` can be created given three integers representing the year, month, and day.

[2] A set of functions allowing a user to examine a `Date`. In this case, functions returning integers representing the year, month, and day are provided, and also two functions returning a character string representations of the `Date`. The `char[]` is a C-style array of characters, `string` is the C++ standard library string type. These functions are marked `const` to indicate that they don't modify the state of the object/variable they are called for.

[3] A set of functions allowing the user to manipulate `Date`s without actually having to know the details of the representation or fiddle with the intricacies of the semantics.

[4] In addition to the explicitly declared operations, `Date`s can be freely copied.

Declared properties are checked at compile time. For example, if a function not declared in class `Date` tries to use a `private` member, that function will cause the compiler to issue an error message. Similarly, the `const` member functions do not modify the state, etc.

Here is a small—and contrived—example of how `Date`s can be used:

```
void f(Date& today)
{
        Date lvb_day = Date(16,dec,today.year());

        if (today.day()==29 && today.month()==feb) {
                // ...
        }

        if (midnight()) today.add_day(1);

        cout << "Today's date:" << today << '\n';
}
```

This assumes that the output operator << has been declared for `Date`s, which we will do in §3.3.

Why is it worthwhile to define an abstract type for something as simple as a date? After all, we could define a structure:

```
struct Date {
        int day, month, year;
};
```

and let programmers decide what to do with that. If we did that, though, every user would either have to manipulate the components of `Date`s directly or provide separate functions for doing so. In effect, the abstraction would be scattered throughout the system, which would make it hard to understand, document, or change. Inevitably, providing a concept as only a simple structure causes extra work for every user of the structure.

Also, even though the `Date` type seems simple, it takes some thought to get right. For example, incrementing a `Date` must deal with leap years, with the fact that months are of different lengths, and so on.

Also, if we ever need to change the representation of `Date` it is useful that the representation be used only by a designated set of functions. For example, if we decided to try representing a `Date` as the number of days before or after January 1 year 0 then only the functions declared in the declaration of `Date` would need changing.

3.1 Defining Member Functions

Naturally, an implementation for each member function must be provided somewhere. For example, here is the definition of `Date`'s constructor:

```
Date::Date(int dd, Month mm, int yy)
{
        y = yy;
        m = mm;
        d = dd;

        int max;

        switch (mm) {
        case feb:
                max = 28+leapyear(yy);
                break;
        case apr:  case jun:  case sep:  case nov:
                max = 30;
                break;
        case jan:  case mar:  case may:  case jul:
        case aug:  case oct:  case dec:
                max = 31;
                break;
        }

        if (d<1 || max<d) throw "bad day";
}
```

The constructor checks that the data supplied denote a valid date. If not, say for `Date(30,Date::feb,1994)`, it throws an *exception*, which indicates in a way that cannot be ignored that something went wrong. For more information about exception handling, see [Stroustrup,1991§9]. If the data supplied are acceptable, the obvious initialization is done. Note that initialization is a relatively complicated operation because it involves data validation. On the other hand, once a `Date` has been created, it can be used and copied without further checking. In other words, the constructor establishes the invariant for the class (in this case, that it denotes a valid date). Other member functions can rely on that invariant and must maintain it. This design technique can simplify code immensely [Stroustrup,1991§12.2.7.1] [Stroustrup,1994§13.2.4].

Most functions are trivial or almost trivial:

```
int Date::day() const
{
        return d;
}

Date& Date::add_year(int n)
{
        if (d==29 && m==feb && !leapyear(y+n)) {
                d = 1;
                m = mar;
        }
        y += n;
        return *this;
}
```

The notation `*this` refers to the object for which a member function is invoked. It is equivalent to Simula's `THIS` and Smalltalk's `self`. Returning a self-reference is a useful convention that allows

chaining of operations. For example:

```
someday.add_day(1).add_month(1).add_year(1);
```

adds a day, a month, and a year to `someday`.

As is common for such simple concrete types, the definitions of member functions vary between trivial and not-too-complicated. Some, such as incrementing a month, are tricky enough to make it worthwhile designing a suitable set of interface functions rather than leaving the manipulation of the data structure completely to users.

3.2 Helper Functions

Typically, a class has a number of functions associated with it that need not be defined in the class itself because they don't need direct access to the representation. For example:

```
int diff(Date a, Date b); // number of days in the range [a,b) or [b,a)
bool leapyear(int y);
Date next_weekday(Date d);
Date next_saturday(Date d);
```

Defining such functions in the class itself would complicate the class interface and increase the number of functions that potentially needed to be examined when a change to the representation is considered.

How are such functions ''associated'' with class `Date`? Traditionally, their declarations were simply placed in the same file as the declaration of class `Date`, and users who need `Dates` would make them all available by ''including'' the file defining the interface, as

```
#include "Date.h"
```

However, this still leaves the association implicit as far as the C++ language rules are concerned, and the names pollute the global name space. A more recent approach is to enclose the class and its helper functions in a namespace:

```
namespace Chrono { // facilities for dealing with time

        class Date {
                // ...
        };

        int diff(Date a, Date b);
        bool leapyear(int y);
        Date next_weekday(Date d);
        Date next_saturday(Date d);

        // ...
}
```

Names from a namespace can be used by explicitly qualifying them with the namespace name or by introducing an alias. For example:

```
void f(Chrono::Date d)
{
        Chrono::Date next_sunday = Chrono::next_saturday(d).add_day(1);
}
```

or

```
using Chrono::Date;  // introduce alias ''Date''

void f(Date d)
{
        using Chrono::next_saturday;  // introduce alias ''next_saturday''

        Date next_sunday = next_saturday(d).add_day(1);
}
```

If detailed control of names is not required, all the names from a namespace can be made available by a single declaration:

```
using namespace Chrono; // make all names from Chrono available

void f(Date d)
{
        Date next_sunday = next_saturday(d).add_day(1);
}
```

Using the more discriminating ways of referring to names in a namespace is less likely to lead to name clashes and surprises [Stroustrup1994§17].

3.3 Overloaded Operators

It is often useful to add functions to enable conventional notation. For example, the `operator==` function defines C++'s equality operator to work for `Date`s.

```
bool operator==(Date a, Date b);  // equality
{
        return a.day()==b.day()
                && a.month()==b.month()
                && a.year()==b.year();
}
```

Other obvious candidates are:

```
bool operator!=(Date, Date);  // inequality
bool operator<(Date, Date);   // less than
bool operator>(Date, Date);   // greater than
// ...

Date& operator++(Date& d)  // increase date by one day
Date& operator--(Date& d); // decrease date by one day

Date& operator+=(Date& d, int n);  // add n days
Date& operator-=(Date& d, int n);  // subtract n days

ostream& operator<<(ostream&, Date d);  // output d
istream& operator>>(istream&, Date& d); // read into d
```

For `Date`, these operators can be seen as mere conveniences, but for many types—such as complex numbers, arrays, and function-like objects—the use of conventional operators is so firmly entrenched in people's minds that their definition is almost mandatory. Usually, it is wise to define only operators that are conventional for a given type and to define them to have their conventional meaning.

When thinking about operator overloading, most people seem primarily to think of arithmetic operators such as `+` and `-`. In our experience, assignment, `=`, subscripting, `[]`, and application, `()` are both more commonly useful and more fundamental to the types for which they are used.

3.4 The Significance of Concrete Classes

We call such simple user-defined types concrete types to distinguish them from abstract types as presented below and to emphasize their similarity to built-in types such as `int` and `char`. They have also been called ''value types,'' and their use ''value-oriented programming.'' Their model of use and the ''philosophy'' behind their design are quite different from what is often advertised as ''object-oriented programming.''

The intent of a concrete type is to do a single relatively small thing well and efficiently. It is not usually the aim to provide the user with facilities to modify the behavior of a concrete type. In particular, concrete types are not intended to display polymorphic behavior (see §5.2).

If you don't like some detail of a concrete type, you build a new one with the desired behavior. If you want to ''re-use'' a concrete type you use it in the implementation of your new type exactly as you would have used an `int`. For example:

```
class Date_and_time {
private:
        Date d;
        Time t;
public:
        Date_and_time(Date d, Time t);
        Date_and_time(int d, Date::Month m, int y, Time t);
        // ...
};
```

The derived class mechanism described in §5.2 can be used to define new types from a concrete class by describing the desired differences, but that implementation technique is beyond the scope of this paper; see [Stroustrup,1994§2.9.1].

A concrete class such as `Date` needs no hidden overhead in time or space. The size of a concrete type is known at compile time so that objects can be allocated on the run-time stack (that is, without free-store operations). The layout of each object is known at compile time so that inlining of operations is trivially achieved. Similarly, layout compatibility with other languages, such as C and Fortran, also comes without special effort.

A good set of such types can provide a concrete foundation for applications. We feel that many programming languages have neglected concrete types. Lack of support for ‘‘small efficient types’’ can lead to gross run-time and space inefficiencies when overly general and expensive mechanisms are used. Alternatively, it can lead to obscure programs and wasted time when programmers are forced to discard expensive abstraction mechanisms in favor of direct manipulation of data structures or lower-level languages.

4 Containers and Generic Programming

As one would expect from a language with a strong emphasis on facilities for designing and using simple types, C++ doesn’t provide sophisticated data structures as built-in types. In this, C++ follows a tradition stretching back to Algol60 of supporting nontrivial concepts, such as input/output, through libraries. Languages following this idea include C, Lisp, Beta, Eiffel, and Smalltalk.

It is not sufficient to provide only simple data structures. Doing that just forces every programmer to reinvent the wheel. Instead, library types must be supplied with the basic operations needed to use them. For example, C doesn’t provide a proper string type. Instead it provides a convention for using arrays of characters and a set of functions for manipulating such strings.

A string is one of the simplest examples of a critically important kind of type, the container. A container is an object used to hold other objects. Other examples are vectors, lists, maps (sometimes called associative arrays and dictionaries), sets, and queues.

In addition to input/output streams and proper character strings, the C++ standard library provides these and other containers [Koenig,1995]. It also provides the basic operations needed to use the containers. These operations—conventionally called algorithms—include sorting, merging, facilities for iterating over containers, facilities for applying operations of elements in containers, etc. The standard library facilities for containers and generic algorithms are derived from Alex Stepanov’s STL library [Stepanov,1994] [Vilot,1994]. This section explores some of the principles behind the STL and some of the techniques used to express them.

4.1 An Elementary Data Structure

A library of fundamental data types is valuable only if the types provided by the library are about as easy to use as built-in types. As an example, we will examine how to handle variable-length arrays first in C and then in C++.

C’s notion of an array matches traditional machine hardware exactly: An array has a fixed size (known at compile time), and it is trivial to obtain pointers to elements of the array. For example:

```
void f1()    /* C or C++ function */
{
#define n 1000
        int squares[n];
        int i;

        for (i = 0; i < n; ++i) squares[i] = i*i;

        /* use squares */
}
```

creates an array containing `n` integer values with indices `0` through `n-1` and sets each element to the square of its index. Unfortunately, the size of such an array, in this case `n`, must be a compile-time constant.

In C, a variable-length array is usually simulated using the library functions `malloc` and `free` that deal in raw memory. For example:

```
void f2(n) int n;   /* a C function */
{
        int *squares = malloc(n * sizeof(int));
        int i;

        for (i = 0; i < n; ++i) squares[i] = i*i;

        /* use squares */

        free(squares);
}
```

To make this work, C supports a form of type punning—it is possible to take an array of one type and treat the memory it occupies as if it really contained memory of another type. This makes it possible to assign the result from `malloc` to `squares`. C's definition of indexing is what makes it possible to refer to `squares[i]` as if it were an element of an array. Probably the greatest inconvenience of using C this way is the requirement to free the memory explicitly when done with it.

Now let us look at how C++ handles variable length arrays. As with built-in arrays, C++ library arrays are one-dimensional. Multi-dimensional arrays are most commonly used for numerical computation, which is supported by a separate numerical library. A one-dimensional array is called a `vector`, and is used something like this:

```
void f3(int n)    // C++ function
{
        vector<int> squares(n);

        for (int i = 0; i < n; ++i) squares[i] = i*i;

        // use squares
}
```

This is not much more difficult than using a built-in array: As for the built-in arrays, there is no special requirement to free the memory used by `squares`; that memory is automatically freed when the variable goes out of scope†.

Making this work for an array size that is not a compile time constant and for an array that is a user-defined type requires the ability

---

† Because `f3()` uses free store and `f1()` uses the stack, `f3()` incurs a fixed allocation overhead, which depends, among other things, on how fast the system's memory allocator is and how much trouble the compiler takes to optimize uses of the standard library. In the (worst and unrealistic) case where *use squares* was nothing, with a compiler that uses the allocator that comes with the machine and no special optimization, we measured the overhead to be to between a factor of 2 and a factor of 3 depending on the size of the vector. On the other hand, when *use squares* was printing out the vector there were no measurable performance difference. We timed an intermediate example, where *use squares* was to take the square root of each element. In this case, the overhead varied from 5% to 58% depending on the size of the vector. We leave it for the reader to decide in which situations the overhead might be significant. There are no significant overhead in `f3()` compared to use of C-style variable length array in `f2()`.

[1] for `vector<int>` to be a distinct type from, say, `vector<float>`;
[2] for the library to define what it means to create an object of type `vector<int>` with a particular parameter;
[3] to define the meaning of subscripting (for example `squares[i]`); it no longer suffices to use the C array/pointer equivalence;
[4] to say what happens when a `vector` goes out of scope.

Indeed, much of the complexity in C++ is there primarily for use by library authors. More language mechanism is needed to allow a library to define useful arrays than would be needed to include a particular definition of arrays in the language itself.

As one would expect from a data structure implemented in a library, more operations are available than just the most elementary ones. For example, suppose we wanted our vector `squares` to contain only values greater than 1000. Taking advantage of our knowledge that we stored values in `squares` in ascending order, we might like to find the first element greater than 1000, copy that one and all the subsequent ones to the beginning, and then shrink the `vector` to the appropriate size. Here is a straightforward way to do that:

```
void g1(vector<int>& squares)
{
        int n = squares.size();

        // find first square greater than 1000:

        int k = 0;
        while (k<n && squares[k]<=1000) ++k;

        // move larger squares:

        int j = 0;
        while (k < n) squares[j++] = squares[k++];

        // resize squares so that its size becomes m
}
```

In addition to missing the part that resizes `square`, this code is tedious and error-prone. What we really want to do is two things:

[1] find the first element, if any, of `squares` that is greater than 1000, and
[2] erase the elements of `squares` before the one we found.

The library offers ways to do that directly. First we write a *predicate* function, which checks if its argument is greater than 1000:

```
bool bigger1000(int n) { return n > 1000; }
```

Next we use a standard library function called `find` to locate the first element for which `bigger1000` is true:

```
void g2(vector<int>& squares)
{
        vector<int>::iterator vi =
                find_if(squares.begin(), squares.end(), bigger1000);

        // resize squares so that its size becomes m
}
```

This last example introduces three things we haven't seen before:

[1] the library defines a type `vector<int>::iterator` that can be used to mark a location in a `vector<int>`;
[2] every `vector` has a pair of member functions called `begin` and `end`, which return iterators that identify the initial element and a point one past the last element of the `vector`; and
[3] the library function `find_if` locates the first element between the points identified by two iterators that satisfies the property given by its third argument. In this case, the third argument is a pointer to

the function `bigger1000()`; `find_if` calls through that pointer to check each element. After calling `find_if`, the vector iterator `vi` will identify either the first element of `squares` that is larger than 1000 or a point one past the end of `squares`. All that is left to do is erase the elements of `squares` starting at the beginning and ending just before `vi`:

```
void g3(vector<int>& squares)
{
        vector<int>::iterator vi =
                find_if(squares.begin(), squares.end(), bigger1000);
        squares.erase(squares.begin(), vi);
}
```

This will work even if `vi` points past the end. Of course, we can combine these two expressions and do away with the local variable `vi`:

```
void g4(vector<int>& squares)
{
        squares.erase(squares.begin(),
                find_if(squares.begin(), squares.end(), bigger1000));
}
```

4.1.1 Library Support for Predicates

For many, `g4()` is as terse, elegant, and efficient as they could wish. However, we can do better still because the library offers some tools for building objects that behave like predicate functions. Let us see how we can eliminate the predicate `bigger1000()` and replace it with standard library features. The point of this is to demonstrate that when the basic library facilities become familiar, it becomes unnecessary to invent tiny functions just to implement trivial predicates.

There is a library type called `greater<int>` whose objects take a pair of integers and determine whether the first is greater than the second. In other words, if we declare

```
greater<int> gt;
```

then `gt(3,4)` would be `false` and `gt(4,3)` would be `true`. These objects are not truly functions, but they act like functions. We therefore call them function objects.

There is also a library function called `bind2nd` that takes a predicate and a value and yields an object that, when called with a single argument, applies the predicate to that argument and the value. This is confusing to describe, but easy to use:

```
(bind2nd(gt, 1000)) (999)
```

is `false` and

```
(bind2nd(gt, 1000)) (1001)
```

is `true`. We can therefore use `bind2nd(gt, 1000)` as our predicate instead of `bigger1000()` when calling `find_if`:

```
void g5()
{
        greater<int> gt;

        squares.erase(squares.begin(),
                find_if(squares.begin(), squares.end(),
                        bind2nd(gt, 1000)));
}
```

Again, we can go further still by eliminating the local variable `gt`. The explicit constructor call `greater<int>()` will serve the same purpose by creating an anonymous object:

```
void g6()
{
        squares.erase(squares.begin(),
                find_if(squares.begin(), squares.end(),
                        bind2nd(greater<int>(), 1000)));
}
```

We can think of the body of this function as meaning

> ''Remove from `squares` all the elements up to and not including the first element that is greater than 1000.''

For programmers without experience with functional languages, this may appear confusing at first glance, but that is mostly because of unfamiliarity. Once one understands what the original operations do, we find this code easier to understand than the original ''straightforward'' version, `g1()`. It is also easier to convince ourselves of its correctness.

Importantly, the notational convenience of `g6()` has not been bought at the cost of run-time inefficiency compared to the conventional C-style version `g1()`†.

4.2 Another Elementary Data Structure

People who mostly use languages that, like C, Fortran, Basic, or Pascal, support arrays more conveniently than lists often use arrays when they really wanted lists. People who mostly use languages that, like Lisp or ML, support lists more conveniently than arrays, are similarly biased toward lists. Our `squares` example, contrived as it is, illustrates an array bias that could be expensive: deleting the initial `k` elements of an `n`-element array will rarely be faster than deleting the initial `k` elements of an `n`-element list and may well be much slower. Suppose we wanted to use lists instead of vectors in our `squares` example. How would we do it?

Using lists in C is so messy that we will leave it to the reader as an exercise. The standard C++ library provides lists that are about as easy to use as arrays. The main difference from the viewpoint of this example is that lists do not offer an indexing operation. We must therefore use one of several available ways of appending an element to a list; the most convenient in the present context looks like this:

```
void fill(list<int>& squares)
{
        for (int i = 0; i<n; ++i) squares.insert_back(i*i);
}
```

Here, `insert_back` is a member function that appends an element to the end of a `list`. This makes it possible to build up a list of `n` elements without having to use indexing. Indeed, class `vector` has a `insert_back()` member function as well, so the loop above would work both for `vector`s and `list`s.

Not only that, but if we want to find the first element of `squares` that is greater than 1000 and remove everything before it, the same expression as before will work here too:

```
void h(list<int>& squares)
{
        squares.erase(squares.begin(),
                find_if(squares.begin(), squares.end(),
                        bind2nd(greater<int>(), 1000)));
}
```

Interestingly, although the expression is the same, many of its components have different types when `squares` is a `list` than they do when `squares` is a `vector`. So, for example, `squares.begin()` yields a value of type `list<int>::iterator` when `squares` is a `list<int>`, the `find_if` function needs to execute completely different code (because accessing the elements of a list is different from accessing the elements of a vector), and so on. It is common for things like this to be possible in languages

---

† Naturally, the performance of such functions is implementation dependent and it is not easy to say what a comparison of such simple examples really means. However, it might be relevant to mention that we measured `g1()` (C-style) to be on average 5% slower than `g4()` (STL Using `bigger1000`) and on average 15% slower that `g6()` (pure STL library). Inlining is the reason that `g6` is faster that `g4()`.

like Lisp or Smalltalk, where types are not determined until execution time, but it is unusual in languages that support strong static typing. What in C++ makes this possible?

4.3 Function Templates and Compile-time Polymorphism

Object-oriented programming is built on top of run-time polymorphism: the ability to choose, during program execution, among functions with similar signatures defined as members of a collection of related types. We will look at that style of programming in more detail in §5. In addition, C++ has function templates, which offer a kind of compile-time polymorphism: every template offers a choice, made during compilation, among operations on types that may be completely unrelated.

Function templates are similar to the generic operations provided in languages like CLU, Modula-3, and Ada. However, they they are unusually flexible in the sense that they work both with built-in and user-defined types and do not require explicit declaration of the types with which they will eventually be used.

Here is a simple example:

```
template<class T> T abs(T x)
{
        if (x < 0) return -x;
        return x;
}
```

The word `class` above simply means ‘‘type:’’ `T` can represent any type, not just a user-defined type. So, for example, `abs(-3)` is `3` (and has type `int`), `abs(-42.1)` is `42.1` (and has type `double`), and so on.

This template defines `abs` for any type `T` that supports copying, unary `-`, and binary `<`. That implies, among other things, that it does not work for complex numbers because they do not define `<`. The result of `abs(z)` where `z` is a `complex` is a compile time error. For reasons like this, it is possible to define template specializations, which work for specific types:

```
double abs(complex z)
{
        return sqrt(pow(re(z),2)+pow(im(z),2));
}
```

This specialization will be called if `abs` is applied to a `complex` argument and the template will be used for other argument types.

With even this little bit of knowledge, it is possible to begin to see how things like `find_if` can be made to work. Consider, for example the following implementation:

```
template<class I, class P> I find_if(I begin, I end, P pred)
{
        while (begin!=end && !pred(*begin)) ++begin;
        return begin;
}
```

This template function says little about the specific types `I` (for *iterator*) and `P` (for *predicate*); in consequence, the function can be used on quite a variety of types.

For example, consider a built-in array:

```
void k()
{
        int a[100];

        int* p = find_if(&a[0], &a[100], bigger1000);
}
```

Here, `bigger1000` is our function from §4.1 that tests if its argument is greater than 1000. The types `I` and `P` are ‘‘pointer to `int`’’ and ‘‘pointer to function taking `int` and returning `bool`,’’ respectively, so in this particular context we could have written `find_if` this way:

```
int* find_if(int* begin, int* end, bool (*pred)(int))
{
        while (begin!=end && !pred(*begin)) ++begin;
        return begin;
}
```

This is just a C program; it does a linear search of the elements of an array in the obvious way.

Now let us look at how we used `find_if` in `h()` where it was used on `squares` and `squares` was a `list<int>`:

```
find_if(squares.begin(), squares.end(), bind2nd(greater<int>(), 1000))
```

Here type `I` is the type of `squares.begin()`. We don't actually know what that type is, but its name is `list<int>::iterator`. All we know beyond its name is that it denotes an element of type `int` somehow. We could think of an iterator as a simple pointer to `int`, though for a `list` a simple `int*` is an unlikely candidate for an iterator type.

Our use of `find_if` is therefore equivalent to what we would have if we wrote it this way:

```
typedef typename list<int>::iterator I;

I find_if(I begin, I end, bool(*pred)(int))
{
        while (begin!=end && !pred(*begin)) ++begin;
        return begin;
}
```

We still don't know what this does. We have, however, reduced the problem of understanding it to a previously unsolved problem, namely understanding how `list<int>::iterator` works. Moreover, if we know that we want this function to do a linear search, we can infer from that the behavior that `list<int>::iterator` must have.

Most fundamental are the facts that `begin` is a `list<int>::iterator` and we pass `begin` as an argument and return it as a result. That means it must support copying. Moreover, comparing `begin` with `end` requires that `list<int>::iterator` must support comparison and, presumably, that the comparison must yield some sensible result. Finally, because we use `*begin` and `++begin`, those operations too must do the right things, whatever those are. If we make the `list<int>::iterator` type do all those things, `find_if` will work.

4.3.1 Iterator Categories

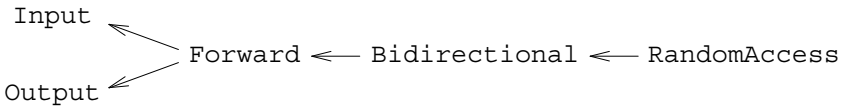
The standard C++ library defines what ''all those things'' are. More specifically, it defines five iterator categories and says what it takes for a type to be a member of each of them. It then says, for the library functions that accept iterators, what category of iterator each one is expected to be.

The simplest kind of iterator is called an input iterator; it does just enough to allow a sequential data structure to be read but not written. Thus, if `p` is an object of an input iterator type, `*p` and `++p` do sensible things, but `--p` might not. You can find the formal definition in the draft ANSI/ISO C++ standard [Koenig,1995] or in [Stepanov,1994].

There are also output iterators, which allow a sequential data structure to be written but not read. The difference between an input and an output iterator is that if `p` is an output iterator, `*p` may only be written but not read.

If a single object can serve both as an input and an output iterator, we call it a forward iterator. A forward iterator that also supports the decrement operator, `--`, is called a bidirectional iterator. Finally, a bidirectional iterator that also supports subscripting and other operations analogous to pointer arithmetic is called a random access iterator. This can be represented graphically:

Iterator categories:



their operations:

```
++ * = == !=                    --                []
```

This iterator nomenclature is not part of the C++ language. Instead, it is part of the standard library documentation. Thus, for example, the description of `find_if` states that the first two arguments must be input iterators that delimit a range of values.

C++ templates do not require the author of functions like `find_if` to declare explicitly that its arguments should be input iterators. In fact, there is no explicit way to declare such things even if the author wanted to. We have heard numerous suggestions that C++ should make it possible to write `find_if` in a style similar to the following:

```
template<class I: input_iterator, class P: predicate>
I find_if(I begin, I end, P pred)
{
        // ...
}
```

Why does C++ offer no such facility? There are three main reasons:

[1] Any such facility would have to take into account not only inheritance but also built-in types and operations on types not defined as members (such as the ‘‘helper functions’’ in §3.2 and §3.3). Ordinary pointers meet the requirements for random-access iterators when they are used to point to elements of (built-in) arrays. That means we would need some way of saying that for any type `T`, `T*` is a random-access iterator. Otherwise, we would have to forego the ability to use functions like `find_if` on built-in arrays.

[2] The facility would offer little additional safety, if any. The main benefit would be that errors would be detected when a template function, such as `find_if`, is called instead of when code is generated for it; we believe that this benefit alone is not enough to justify a whole new type-checking facility.

[3] Even if such a facility existed and checked usage completely at the earliest possible instant, that would still not guarantee safety. To work correctly, a template requires that its parameter type provide the expected operations with the expected semantics. Specifying ‘‘the expected operations’’ can be messy and constraining. Specifying ‘‘the expected semantics’’ can be surprisingly difficult. For example, most attempts to specify something as simple as a less than operator, `<`, in general can involve the programmer in the intricacies of the IEEE floating-point value `NaN` (not a number). We prefer to leave such complexity in the documentation.

In general, we know of no way of expressing constraints on template parameters that wouldn’t be either too cumbersome or too constraining [Stroustrup,1994,§15.4]. Instead, C++ provides mechanisms for providing separate implementations, called specializations, for special cases. For example, in addition to providing a general `list` template, one can provide versions to be used for lists of pointers (in general), and for lists of `void*` (in particular).

4.4 Strategy, Style, and Interface Conversion

Users rely on library code. Conversely, libraries often have to rely on user code for critical operations done to user data. Examples are copy operations for objects passed to a container, compare functions passed to a sort routine, and a user-defined class overriding a virtual draw function in a graphics class.

Over the years, it has become obvious that techniques making such dependencies rely less on specific names and interface styles significantly increases the flexibility and usefulness of libraries. The template mechanism has played a key role in such tayloring of interfaces.

4.4.1 Minimizing Run-time Resolution

Flexibility is often achieved by postponing decisions until run time. Sometimes, that is just right, but at other times the convenience is bought at a cost in speed and safety. Consider the well known C (and C++) standard library function `printf()`:

```
#include <stdio.h>

main()
{
        printf("%s", "Hello world\n");
}
```

Here, `printf()` determines the type of its second argument at run time. In general, it has to, because its first argument, the format string, might be a variable. In most cases, static type checking of `printf()` is possible. However, from an implementer's viewpoint, it is easier to put this kind of run-time type checking into the `printf` library function than into the compiler.

The C++ equivalent,

```
#include <iostream.h>

main()
{
        cout << "Hello world\n";
}
```

does not rely on run-time typing. Instead, the types of `cout` and of the the string literal are used to select during compilation the appropriate version of the `<<` operator to use. This means that there is no run-time overhead involved in finding the right kind of output conversion to use and no possibility that the wrong choice will cause a crash.

The cooperation between the user and the library is established through the convention that the `<<` operator is used for output. If a library or a user needs to support output of a new type, a new `<<` is provided.

4.4.2 Templates for Interface Conversion

Sometimes conventions clash. For example, there may be one well-established convention for I/O and another for container interfaces. Consequently, it can be useful for library routines *not* to rely directly on the interfaces of the objects they use. Instead they rely on auxiliary objects that express the mapping between the expectation of library code and the user code interfaces. For example, the algorithms in the STL library doesn't use containers directly. Instead they access their input and output through iterators.

This strategy makes it possible to write iterator classes whose sole purpose is to impose a particular interface on objects of some class that already exists. For example, many popular algorithms read their input one element at a time from a source. It makes perfect sense to let those algorithms get their input from an input stream. In fact, anything else requires clumsy workarounds. Consequently, the standard C++ library offers a template class called `istream_iterator`. Each object of that class obeys the rules for input iterators, but such objects do not iterate over a data structure in the ordinary sense. Instead, an `istream_iterator` yields, in turn, each of the data values in a particular input stream, read according to the usual rules for the `>>` operator.

For example, suppose we say

```
input_iterator<string> ins(cin);
```

Then `ins` is an object that on request will read strings from `cin`, so that if `s` is a `string`,

```
s = *ins++;
```

has the same effect as

```
cin >> s;
```

The STL model requires that we iterate from somewhere to somewhere. Consequently, we need a value indicating ''end of file'' that we can compare the iterator `ins` to. Such a value is used by default for an uninitialized `input_iterator<string>`, so that we can say something like this:

```
input_iterator<string> ins(cin);
input_iterator<string> eof;

void f1()
{
        while (ins != eof) {
                s = *ins++;
                // ...
        }
}
```

and the loop will be executed once for each `string` in the standard input file.

This is equivalent to:

```
void f2()
{
        while (cin >> s) {
                // ...
        }
}
```

However, defining `input_iterator` makes it possible for the algorithm library to use the input/output stream library unmodified. For example we can read all the strings in the standard input into a `vector<string>` without writing an explicit loop. Instead, we can create the `vector` directly from the standard input:

```
vector<string> vs(ins, eof);
```

Here, `vs` is constructed with two arguments, both iterators; doing that causes `vs` to be initialized with a copy of the elements in the range delimited by those iterators. In this case, that range is the entire contents of the standard input file.

Along similar lines, we can create an output iterator attached to the standard output file:

```
ostream_iterator<string> outs(cout, "\n");
```

Here, the second argument to the `ostream_iterator` constructor is a string that will be written after each use of the `ostream_iterator`. Thus, for example

```
*outs++ = "Hello world";
```

will print `Hello world` followed by a newline character.

With these iterators, we can read all the strings in the standard input and print them on the standard output this way:

```
void g1()
{
        vector<string> vs(ins, eof);
        copy(vs.start(), vs.end(), outs);
}
```

We can even write

```
void g2()
{
        copy(ins, eof, outs);
}
```

which would write each string as soon as it read it.

However, reading the entire input before producing output makes it possible to do interesting things before printing, such as sorting the elements. For example, this complete program (except for including header files) sorts standard input onto its standard output:

```
int main()
{
        istream_iterator<string> ins(cin), eof;
        ostream_iterator<string> outs(cout, "\n");

        vector<string> vs(ins, eof);
        sort(vs.begin(), vs.end());
        copy(vs.begin(), vs.end(), outs);

        return cout && cin; // use state of streams as result
}
```

Of course, C++ provides a form of run-time polymorphism as well, which is the subject of the next section.

5 Design of Class Hierarchies

From Simula, C++ borrowed the concept of a class as a user-defined type and the concept of class hierarchies. In addition, C++ borrowed the idea for system design that classes should be used to model concepts in the programmer's and the application's world. This is often called object-oriented design and is the key to effective use of classes. Language constructs directly support these design notions; the application of design concepts is what distinguishes effective use of C++ from simpler uses of the language constructs as notational props for more traditional types of programming.

A concept doesn't exist in isolation. For example, try to explain what a car is. Soon you'll have introduced the notions of wheels, engines, drivers, pedestrians, trucks, ambulances, roads, oil, speeding tickets, motels, etc. Consequently, when we try to map concepts into classes, we soon find the need to express relationships between classes. However, we can't express arbitrary relationships directly, and even if we could we wouldn't want to. Our classes should be more narrowly defined than our everyday concepts, and more precise. Languages that borrow from Simula are particularly adept at expressing hierarchical relationships between classes.

5.1 Class Hierarchies

Consider a simple design problem: Provide a way for a program to get an integer value from a graphical user interface. This can be done in a bewildering number of ways. To insulate our program from this variety, and also to get a chance to explore the possible design choices, let us start by defining our program's model of this simple input operation. We will leave the details of implementing it using a real user-interface system for later.

The idea is to have a class `ival_box` that knows what range of input values it will accept. A program can ask an `ival_box` for its value, and ask it to prompt the user if necessary. In addition, a program can ask an `ival_box` if a user has changed the value since the last operation initiated by the program.

Because there are many ways of implementing this basic idea, we must assume that there will be many different kinds of `ival_box`es, such as sliders, plain boxes where a user can type a number, dials, voice interaction, and so on.

5.2 A Traditional Class Hierarchy

Our first solution will be a traditional class hierarchy as commonly found in Simula or Smalltalk programs.

Class `ival_box` defines the basic interface to all `ival_box`es and specifies a default implementation that more specific kinds of `ival_box`es can override with their own versions. In addition, we declare the data needed to implement the basic notion.

```
class ival_box {
protected:
        int val;
        int low, high;
        bool changed;
public:
        ival_box(int ll, int hh)
                { changed = false; low=ll; high=hh; val = ll; }
        virtual int get_value()
                { changed = false; return val; }
        virtual void set_value(int i)
                { changed = false; val = i; }
        virtual void prompt()
                { }
        virtual bool was_changed() const
                { return changed; }
};
```

The default implementation of the functions is pretty sloppy and provided here primarily to illustrate the intended semantics. A realistic class would, for example, provide some range checking.

Given this basic definition of `ival_box`, we can derive variants of the concept from it. For example:

```
class ival_slider : public ival_box {
        // graphics stuff to define what the slider looks like, etc.
public:
        ival_slider(int, int);

        int get_value();
        void prompt();
        bool was_changed();
};
```

A class like `ival_slider` is said to be *derived* from class `ival_box` and `ival_box` is said to be a *base* of `ival_slider`. Alternatively, we can call `ival_box` the superclass of `ival_slider` and `ival_slider` a subclass of `ival_box`. The notation

```
class ival_slider : public ival_box { /* ... */ };
```

defines `ival_slider` to be a subtype of `ival_box`. In other words, we can manipulate an `ival_slider` as we would an `ival_box`; they share the interface defined by `ival_box`.

Further, a `virtual` function in a base class, say `ival_box::prompt()`, can be overridden by defining a function with the same name and the same argument types in a derived class, say `ival_slider::prompt()`. That done, a call of `prompt()` on an object of the derived class will invoke ‘‘the right’’ function, even if the variable used to refer to the `ival_slider` is a plain pointer to `ival_box`. For example:

```
void f()
{
        ival_box* p = new ival_slider(10,700);
        p->prompt();
}
```

Here, the initialization of `p` is legal because `ival_slider` is a subtype of `ival_box`, and the call of `prompt()` will invoke `ival_slider::prompt()` because `prompt()` is a virtual function overridden in `ival_slider`.

Getting ‘‘the right’’ behavior from `ival_box`’s functions independently of the exact kind of `ival_box` actually used is called polymorphism. A type with virtual functions is called a polymorphic type. To get polymorphic behavior in C++, objects must be manipulated through pointers or references. The reason for this is that when manipulating an object directly, its type is always known during compilation so that run-time polymorphism is not needed.

The `new` operator creates an object of a given type on the free store, initializes it by an invocation of the

appropriate constructor, and returns a pointer to the resulting object .

A derived class constructor need not take the same set of arguments as its base class. Typically a derived class has its own distinct requirements for arguments relating to its particular variant of the idea represented by the base class. However, to simplify the discussion here, we will define all of our constructors to take two `int`s bounding the desired range.

The data members of `ival_box` were declared `protected` to allow access from derived classes. Thus, `ival_slider::get_value()` can deposit a value in `ival_box::val`. A `protected` member is accessible from a class' own members and members of derived classes, but not to general users.

In addition to `ival_slider`, we would define other variants of the `ival_box` concept such as `ival_dial` where you select a value by turning a knob, `flashing_ival_slider` that flashes when you ask it to `prompt()`, and `popup_ival_slider` that responds to `prompt()` by appearing in some prominent place where it is hard for the user to ignore.

A programmer might use these ''`ival` classes'' like this:

```
void interact(ival_box* pb)
{
        pb->prompt();   // alert user
        // ...
        int i = pb->get_value();
        if (pb->was_changed()) {
                // new value; do something
        }
        else {
                // old value was fine; do something else
        }
        // ...
}

void some_fct()
{
        ival_box* p1 = new ival_slider(0,5);
        // ...
        interact(p1);
        ival_box* p2 = new ival_dial(1,12);
        // ...
        interact(p2);
}
```

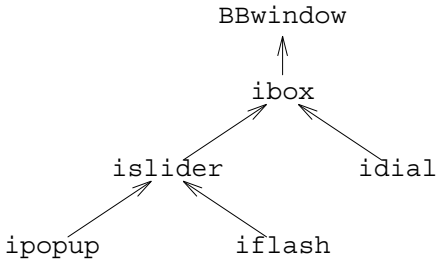
Note that most application code is written in terms of (pointers to) plain `ival_box`es the way `interact()` is. That way, the application doesn't have to know about the potentially large number of variants of the `ival_box` concept. The knowledge of such specialized classes is isolated in the relatively few functions that create such objects. This isolates users from changes in the implementations of the derived classes and most code can be oblivious to the fact that there are different kinds of `ival_box`es.

Where would we get the graphics stuff from? Most user-interface systems provide a class defining the basic properties of being an entity on the screen, so if we use the system from ''Big Bucks Inc.'' we would have to make each of our `ival_slider`, `ival_dial`, etc., classes a kind of `BBwindow` class. This would most simply be achieved by rewriting our `ival_box` so that it derives from `BBwindow`. That way, all our classes inherit all the properties of a `BBwindow`. For example, every `ival_box` can be placed on the screen, obey the graphical style rules, be resized, be dragged around, etc., according to the standard set by the `BBwindow` system. Our class hierarchy would look like this:

```
class ival_box : public BBwindow { /* ... */ };    // rewritten

class ival_slider : public ival_box { /* ... */ };

class ival_dial : public ival_box { /* ... */ };

class flashing_ival_slider : public ival_slider { /* ... */ };
```

```
class popup_ival_slider : public ival_slider { /* ... */ };
```

or graphically using obvious abbreviations:



5.2.1 Critique

This design works well in many ways, and for many problems this kind of hierarchy is a good solution. However, there are some awkward details that could lead us to look for alternative designs.

We retrofitted `BBwindow` as the base of `ival_box`. This is not quite right. The use of `BBwindow` wasn't part of our basic notion of an `ival_box`; it was an implementation detail. Deriving `ival_box` from `BBwindow` elevated an implementation detail to a first-level design decision. That can be right, say when working in the environment defined by ''Big Bucks Inc.'' is a key decision of how our organization conducts its business. However, what if we also wanted to have implementations of our `ival_box`es for systems from ''Imperial Bananas,'' ''Liberated Software,'' and ''Compiler Wizzes?'' This would require us to write and maintain four distinct versions of our program:

```
// BB version:

        class ivalue_box : public BBwindow { /* ... */ };

// CW version:

        class ivalue_box : public CWwindow { /* ... */ };

// IB version:

        class ivalue_box : public IBwindow { /* ... */ };

// LS version:

        class ivalue_box : public LSwindow { /* ... */ };
```

This could become a version-control nightmare.

Another problem is that every derived class shares the basic data declared in `ival_box`. That data is, of course, an implementation detail that crept into our `ival_box` interface also. From a practical point of view, it is also the wrong data in many cases. For example, an `ival_slider` doesn't need the value stored specifically. It can easily be calculated from the position of the slider when someone executes `get_value()`. In general, keeping two related, but different, sets of data is asking for trouble. Sooner or later someone will get them out of sync. Also, experience shows that novice programmers tend to mess with protected data in ways that are unnecessary and cause maintenance problems. Data are better kept private so that writers of derived classes cannot mess with them. Better still, data should be in the derived classes where they can be defined to match requirements exactly and cannot complicate the life of unrelated derived classes. In almost all cases, a protected interface should contain functions, types and constants only.

Deriving from `BBwindow` gave the benefit of making the facilities provided by `BBwindow` available to users of `ival_box`. Unfortunately, it also means that changes to class `BBwindow` may force users to recompile or even rewrite their code to recover from such changes. In particular, the way most C++ implementations work implies that a change in the size of a base class requires a recompilation of all derived classes.

Finally, our program may have to run in a mixed environment where windows of different user-interface systems coexist. This could happen either because two systems somehow share a screen, or because our program needs to communicate with users on different systems. Having our user-interface systems ''wired in'' as the one and only base of our one and only `ival_box` interface just isn't flexible enough to handle that.

5.3 Abstract Classes

So, let's start again and build a new class hierarchy that solves the problems presented in the critique of the traditional hierarchy. That is:

[1] The user-interface system should be an implementation detail that is hidden from users who don't want to know about it.

[2] The `ival_box` class should contain no data.

[3] No re-compilation of code using the `ival_box` family of classes should be required after a change of the user-interface system.

[4] `ival_box`es for different interface systems should be able to coexist in our program.

Several alternative approaches can achieve this. We will present one that maps cleanly into the C++ language.

First we specify class `ival_box` as a pure interface:

```
class ival_box {
public:
        virtual int get_value() = 0;
        virtual void set_value(int i) = 0;
        virtual void prompt() = 0;
        virtual bool was_changed() const = 0;
        virtual ~ival_box() { }
};
```

This is much cleaner that the original declaration of `ival_box`. The data is gone, and so are the simplistic implementations of the member functions. Gone too is the constructor, because there is no data for it to initialize.

Instead, two things have been added. We will describe the role of the function called `~ival_box()`, the destructor, below. The curious `=0` syntax says that a function is a pure virtual function. A class with one or more pure virtual functions is called an abstract class. Because objects of abstract classes cannot be created, pure virtual functions need not be defined. Only objects of non-abstract derived classes can be created; those classes must define all the pure virtual functions they inherit. For example, if we assume that `ival_slider` is derived from `ival_box` and defines all the pure virtuals:

```
void f()
{
        ival_box b1;                        // error: abstract class
        ival_box* p1 = new ival_box;        // error: abstract class
        ival_slider b2;                     // ok
        ival_box* p2 = new ival_slider;     // ok
}
```

The definition of `ival_slider` might look like this:

```
class ival_slider : public ival_box, protected BBwindow {
        // data needed for slider

protected:
        // functions overriding BBwindow virtual functions
        // e.g. BBwindow::draw(), BBwindow::mouse1hit()
```

```
public:
        ival_slider(int,int);
        ~ival_slider();

        int get_value();
        void set_value(int i);
        void prompt();
        bool was_changed() const;
};
```

Interestingly, this declaration allows application code to be written exactly as in the `interact()` and `some_fct()` example above. All we have done is to restructure the implementation details in a more logical way.

The virtual function `ival_box::~ival_box()` and its overriding function `ival_slider::~ival_slider()` are destructors, that is, functions that are implicitly called when an object is destroyed (goes out of scope, is explicitly deleted, etc.). Many classes require some form of cleanup for an object before it goes away. Since the abstract class `ival_box` cannot know if a derived class requires such cleanup, it must assume that it does. Defining a virtual destructor in the base ensures proper cleanup. For example:

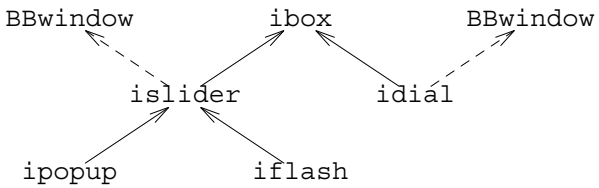
```
void f(ival_box* p)
{
        // ...
        delete p;
}
```

The `delete` operator explicitly destroys the object pointed to by `p`. We have no way of knowing exactly which class the object pointed to by `p` belongs to, but thanks to `ival_box`'s virtual destructor, proper cleanup as (optionally) defined by that class' destructor will be called.

The `ival_box` hierarchy can now be defined like this:

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

class ival_slider : public ival_box, protected BBwindow { /* ... */ };

class ival_dial : public ival_box, protected BBwindow { /* ... */ };

class flashing_ival_slider : public ival_slider { /* ... */ };

class popup_ival_slider : public ival_slider { /* ... */ };
```

or graphically using obvious abbreviations:



Each derived class inherits an abstract class (for example, `ival_box`) requiring it to implement the base class' pure virtual functions, and a `BBwindow` provides them with the means of doing so. Since `ival_box` provides the interface for the derived class, it is publicly derived using `:public`. Since `BBwindow` is only an implementation aid, it is derived using `:protected`. This implies that a programmer using, say `ival_slider`, cannot directly use facilities defined by `BBwindow`; only the interface inherited by `ival_box` and possibly augmented by `ival_slider` is available.

Deriving directly from more than one class is usually called multiple inheritance. Note that `ival_slider` must override functions from both `ival_box` and `BBwindow` so it must be defined by deriving it directly or indirectly from both. As shown in §4.2, deriving `ival_slider` indirectly from `BBwindow` by making `BBwindow` a base of `ival_box` is possible, but has undesirable side effects.

This design is cleaner and more easily maintainable than the traditional one—and no less efficient. It still fails to solve the version control problem, though:

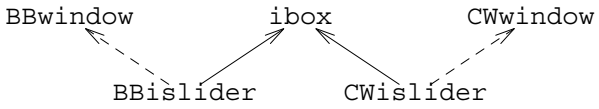
```
// common:

        class ival_box { /* ... */ };

// BB version:

        class ival_slider : public ival_box, protected BBwindow
        { /* ... */ };

// CW version:

        class ival_slider : public ival_box, protected CWwindow
        { /* ... */ };

// ...
```

In addition, there is no way of having an `ival_slider` for `BBwindows` coexist with an `ival_slider` for `CWwindows` even if the two user-interface systems can themselves coexist.

The obvious solution is to define several different `ival_slider` classes with separate names:

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

class BB_ival_slider : public ival_box, protected BBwindow { /* ... */ };

class CW_ival_slider : public ival_box, protected CWwindow { /* ... */ };

// ...
```

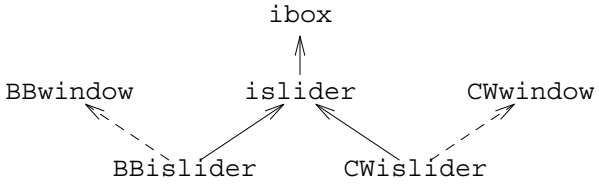
or graphically:



To further insulate our application-oriented `ival_box` classes from implementation details, we can go one step further and first derive an abstract `ival_slider` class from `ival_box` and then derive the system specific `ival_sliders` from that:

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

class ival_slider : public ival_box { /* ... */ };

class BB_ival_slider : public ival_slider, protected BBwindow { /* ... */ };

class CW_ival_slider : public ival_slider, protected CWwindow { /* ... */ };

// ...
```

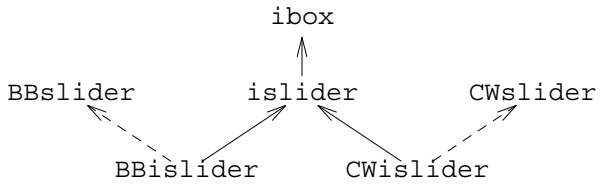
or graphically:



Usually, we can do better yet by utilizing more specific classes in the implementation hierarchy. For example, if the Big Bucks Inc. system has a slider class, we can derive our `ival_slider` directly from the `BBslider`:

```
class BB_ival_slider : public ival_slider, protected BBslider { /* ... */ };

class CW_ival_slider : public ival_slider, protected CWslider { /* ... */ };
```

or graphically:



This improvement becomes significant where—as is not uncommon—our abstractions are not too different from the ones provided by the system used for implementation. In that case, programming reduces to mapping between similar concepts. Derivation from general base classes, such as `BB_window`, is then done only rarely.

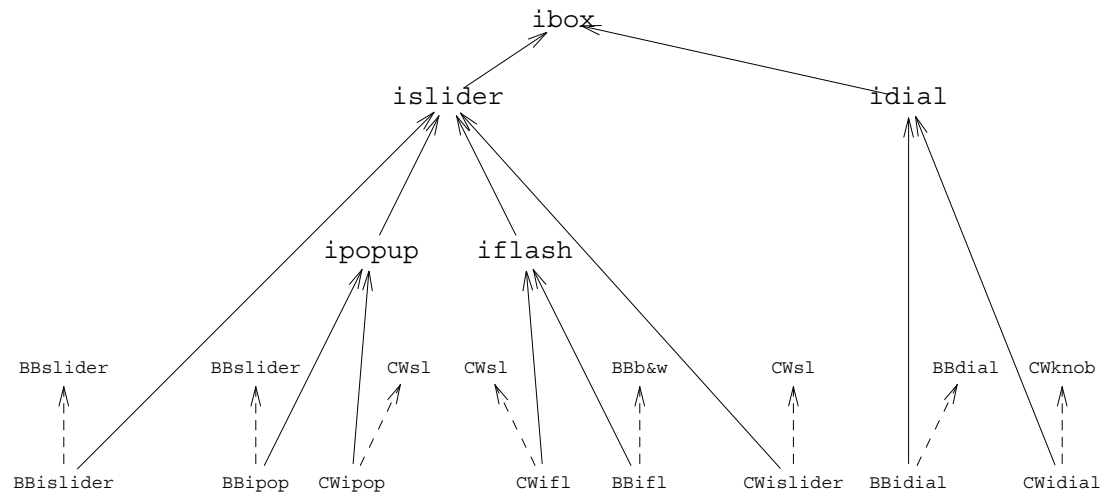
The complete hierarchy will consist of our original application-oriented conceptual hierarchy of interfaces expressed as derived classes:

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

class ival_slider : public ival_box { /* ... */ };

class ival_dial : public ival_box { /* ... */ };

class flashing_ival_slider : public ival_slider { /* ... */ };

class popup_ival_slider : public ival_slider { /* ... */ };
```

followed by the implementations of this hierarchy for various windows systems expressed as derived classes:

```
// BB implementations:

        class BB_ival_slider
                : public ival_slider, protected BBslider { /* ... */ };

        class BB_ival_dial : public ival_box, protected BBdial { /* ... */ };

        class BB_flashing_ival_slider
                : public ival_slider,
                  private BBwindow_with_bells_and_whistles { /* ... */ };

        class BB_popup_ival_slider
                : public ival_slider, protected BBslider { /* ... */ };

// CW implementations:

        class CW_ival_slider
                : public ival_slider, protected CWslider { /* ... */ };

        class CW_ival_dial : public ival_dial, protected CWknob { /* ... */ };

        // ...

// ...
```

or graphically:



Note how the original `ibox` class hierarchy appears unchanged, but is surrounded by implementation classes.

5.3.1 Critique

The abstract class design is flexible, and almost as simple to deal with as the equivalent design relying on a common base defining the user-interface system. In the latter design, the windows class is the root of a tree. In the former, the original application class hierarchy appears unchanged as the root of classes that supply its implementations. In either case, you can look at the `ival_box` family of classes without bothering with the window-related implementation details most of the time.

In either case, the complete implementation of each `ival_box` class must be rewritten when the public interface of the user-interface system changes. However, in the abstract class design almost all user code is protected against changes to the implementation hierarchy and require no recompilation.

5.4 Localizing Object Creation

The flexibility of the abstract class design causes one problem, though. Most of an application can be written using the `ival_box` interface. Further, should the derived interfaces evolve to provide more facilities than plain `ival_box`, then most of an application can be written using the `ival_box`, `ival_slider`, etc., interfaces. However, the creation of objects must be done using implementation-specific names such as `CW_ival_dial` and `BB_flashing_ival_slider`. We would like to minimize the number of places such specific names occur, and object creation is hard to localize unless you do it systematically.

As usual, the solution is to introduce an indirection. This can be done in many ways, but here is a simple one:

```
class ival_box_maker {
public:
        virtual ival_slider* ival_slider(int, int) =0;
        virtual ival_dial* ival_dial(int, int) =0;
        virtual popup_ival_slider* popup_ival_slider(int, int) =0;
        // ...
};
```

For each interface from the `ival_box` family of classes a user should know about, class `ival_box_maker` provides a function making an object. We now represent each user-interface system by a class derived from `ival_box_maker`:

```
class BB_maker : public ival_box_maker {
public:
        ival_slider* ival_slider(int, int);
        ival_dial* ival_dial(int, int);
        popup_ival_slider* popup_ival_slider(int, int);
        // ...
};

class LS_maker : public ival_box_maker {
public:
        ival_slider* ival_slider(int, int);
        ival_dial* ival_dial(int, int);
        popup_ival_slider* popup_ival_slider(int, int);
        // ...
};
```

Each function simply creates an object of the desired interface and implementation type. For example:

```
ival_slider* BB_maker::ival_slider(int a, int b)
{
        return new BB_ival_slider(a,b);
}
```

Given a pointer to a `ival_maker`, a user can now create objects without having to know exactly which user-interface system is used. For example:

```
void f(ival_maker* pim)
{
        // ...
        ival_box = pim->ival_slider(-99,99);
                // instead of new BB_val_slider(-99,99);
                //         or new LS_val_slider(-99,99);
                //         or ...
        // ...
}

BB_ival_maker BBim;
LS_ival_maker LSim;

void g()
{
        f(&BBim);  // let f use BB
        f(&LSim);  // let f use LS
}
```

This technique appears in [Gamma,1994] as the *abstract factory* pattern.

6 C++ Style

C++ is often inaccurately described as an object-oriented language, and (therefore?) often criticized for not fulfilling everybody's fantasies of what an object-oriented language ought to be.

If we *have* to stick a pretentious-sounding label on C++ it must be: C++ is a multi-paradigm language. It supports several styles of programming and combinations of those styles. The traditional summary is [Stroustrup,1994]:

> C++ is a general-purpose programming language that
> – is a better C
> – supports data abstraction
> – supports object-oriented programming

However, the exact scope of this isn't easy to pin down to a simple slogan such as ''Everything is an Object!'' or ''No side effects!'' Such slogans are certainly not among the ideals of C++ even though

support for both object-oriented programming and functional styles of programming is.

Good C++ style is pragmatic, has evolved from the Simula ideas of object-oriented design as modelling, places a premium on direct expression of ideas, shares much of C's concern for low-level efficiency, and is aimed at solving current everyday problems.

Naturally, this is just our view. Nothing is universally held in a community as large as the C++ user community, but our view is directly reflected in the design of C++ [Stroustrup,1994,§4]. Fortunately for people who hold other views, one of our strongest held opinions is exactly that C++ should support a variety of styles. Thus, even though we don't try to provide direct support for every style of programming in C++, we don't go out of our way to prevent styles we don't like, either. Indeed, it is often a source of enjoyment to see people using C++ in ways we did not anticipate—especially when it is successful.

Unfortunately—or maybe fortunately—style is hard to define and must be taught (and learned!) with liberal use of examples. We have presented three areas where C++ provides direct support, where a definite view of design can guide the programmer, and where the design views and resulting coding style reflects experience with C++. The examples were chosen to demonstrate areas that are not universally well-covered by modern programming languages and where current practice—in C++ and other languages—often diverges from our ideal. Thus the examples from §3, §4, and §5 can serve as discriminating cases and possibly as inspiration to do as well or better.

Clearly, by ''style'' we just don't mean rules for indentation of code, the naming of variables, and the banning of unfashionable language features. Good programs are the result of a focus on concepts and sound notions of design, rather than mechanistic language-technical issues. Such issues matter, but at a much more detailed level.

C++ supports enough data abstraction to make it possible to program at as high a level as in many more ''advanced'' languages. Doing so usually requires extensive work designing, implementing and tuning a library supporting the style. Building such a framework should not be everyday work for most C++ programmers. For example, the STL wasn't easy to design (Alex Stepanov and his colleagues worked on the basic ideas for over a decade), was somewhat easier to implement (the current version was about two years of work for two people), and it is quite simple to teach and use.

This is a key idea: first a relatively small group of people develops a library supporting an application domain well. After that, many more people can use the library to develop applications or the next level of library. We are not making a value judgement about programmers here. It easier to use a well-designed library than it is to design and implement it, and the subset of C++ needed to produce a complete, efficient, and elegant library is far larger than what is needed to use it. This has led some people to propose a class system of programmers with the best programmers focused on library development and the worst restricted to application development.

However, the demands on a programmer's skills are a function of both the inherent difficulty of the application and the quality of tools available for its development. Therefore, one cannot blindly assume that lesser skills or fewer language features are needed for application development. Sometimes, things seem the other way around with the library developers benefitting from a relatively limited and well-defined problem domain, and the application developers suffering from being lost in an overly large and complicated design space. From this observation comes the notion that the best way to make progress on a large system is to focus on the development of several libraries or frameworks and then build the system incrementally from those.

The unit of design is not the individual class—in C++ or in any other language. It is a set of classes related by some logical criteria [Stroustrup,1991§12.11.3.3]. For example, the power of the STL comes from the unifying criteria for what constitutes a container, an iterator, etc. Similarly, the discussion of design issues relating to the input operation in §5 would have been impossible had we tried to consider the problem one isolated class at a time.

Another key observation is that not every class is supposed to be used in the same way or obey the same simple-minded design criteria. Often, simplified design rules of thumb are advertised as universal principles and a curious form of reductionism takes the place of calm thinking. Thus, we find people arguing that because some classes are best designed as part of a hierarchy, every class must be designed to be part of a class hierarchy; that because it makes sense for some functions to be `virtual`, every function must be `virtual`; and that because some interfaces are best described as abstract classes, no class presented to a users may contain data.

This kind of purely language-driven thinking makes no sense to us. We must focus on the concepts in the application and map them into the language constructs in the most appropriate way. In other words, we must design first and keep our programming-language-technical concerns secondary. On the other hand, we don't consider totally language-independent design practical. The design must map into the language used for its implementation in a way that suits the fundamental structure of the language. In particular, a design for a C++ program that tries to subvert C++'s static type system will be ugly, unpleasant to implement, and hard to maintain. Against the fundamental structure of a language—any language—one can win Pyrrhic victories only.

One implication of this is that major interfaces are usually best defined in terms of specific user-defined types and that a class should provide an interface that match a single coherent concept. This allows better type checking, and wherever possible static (compile time) checking should be used to minimize confusion, run-time errors, and the need for run-time checking of arguments passed across an interface. The `Date` constructor can be used to illustrate some tradeoffs:

```
Date::Date(int d, Month m, int y);
```

`Month` is a user-defined type (an enumeration), so we can't get much confusion from that. People reading the declaration know what is expected; should they nevertheless mess up, the compiler catches the problem:

```
Date d1(1978,2,21);          // error: 2 is not a Month
Date d2(1978,Date::feb,21);  // ok
```

However, we reversed the year and the day. The `Date` constructor's check of the range of dates in February will catch that at run-time.

Had `Date` been critical in our design, we might have introduced a `Day` or a `Year` type to allow stronger compile-time checking. For example:

```
class Year {
        int y;
public:
        explicit Year(int i) { y = i; }     // construct Year from int
        operator int() const { return y; }  // conversion: Year to int
};

class Date {
        Date(int d, Month m, Year y);
        // ...
};

Date d3(1978,feb,21);       // error: 21 is not a Year
Date d4(21,feb,Year(1978)); // ok
```

The `Year` class is a simple ''wrapper'' around an `int`. Thanks to the `operator int()` a `Year` is implicitly converted into an `int` wherever needed. Thanks to the `explicit` constructor, an `int` can be explicitly (only) converted into a `Year`. Because `Year`'s member functions are easily inlined, no run-time or space costs are added. This reflects the rule for the design of C++ that to be useful, a facility mustn't just be elegant: it must also be affordable in real programs. We don't have any strong rules for where such added compile time checking is worth the added effort from the programmer. If necessary, such a simple wrapper class can contain additional run-time checks.

Letting a class represent a single coherent concept (only) tends to lead designs away from hierarchies based on very general base classes. This is good because over time such base classes tend to acquire data and functions to the point where they become a burden. The classic example is a base class for a container hierarchy. Such a class tends to provide a superset of the operations needed for individual containers. It may, for example, provide operations for indexing, list operations, size adjustments, access to associative data structures, etc. Because not every specific container can implement every operation on the ''fat'' container interface, inefficiencies, run-time checking, and bugs result; see also [Stroustrup.1991,§13.6]. A clean C++ program tends to be a forest of classes rather than a single large tree.

Naturally, *many* C++ designs violate one or more of our suggested rules. This is partly because not everybody agrees about these design rules, and partly because of inexperience about design in the C++

community. There may very well be more good designers in the C++ community than in any other programming community, but there certainly are more novices. The rapid growth of C++ usage ensures that. We can teach design to small groups, and even to larger organizations. However, getting design technique applied on a large scale (hundreds or thousands of programmers) is a task no language community has been spectacularly successful at—yet.

7 Sociological Observations

A programming language by itself is useless. Unless supported by tools, techniques, and a user community, a language is simply an intellectual plaything. There is a need for experimental languages, niche languages, languages devoted to the pursuit of beauty without compromise. However, C++ was never meant to be one of those; it was designed and evolved to be a practical tool.

Like the success of C, the success of C++ was no accident. Naturally, a certain element of good fortune was involved in both cases; nothing succeeds on a large scale without a bit of luck. However, a large part of that success came from an effort to make C++ the best language possible, rather than the best possible language.

Throughout its evolution, C++ was heavily influenced by a desire to make it a useful tool to a community of potential users who already existed, whose problems we knew reasonably well. Another important aspect was restraint: C++ was not allowed to grow without solid feedback on what we already had, without practical experience with problem areas (where what we had felt ‘‘not good enough’’), and without concerns for compatibility and transition issues. Theory was never a sufficient reason for adding something to C++. Theory determines the form of what is added but not what is needed.

7.1 The best language possible

What is the best programming language? We have lost count of the number of times we have heard that question asked and answered. Most of those questions and answers have little meaning because they focus on language-technical issues to the exclusion of vitally important aspects of how a non-experimental programming language is used: programming languages, like other tools, are useful only in context.

A context has several parts. For example:

[1] What problems, or kinds of problems, will the language be used to solve?
[2] What skills do the people have who might use the language? Which new skills are they able and willing to acquire?
[3] What languages are available, or can be made available? What is the cost of making them available?
[4] How easy is it to obtain access to experts who can help answer the questions that inevitably arise?
[5] Are useful libraries available?
[6] Is it necessary for programs to work with other programs that already exist?
[7] What are the performance constraints?
[8] Will it become necessary to run one’s programs on other machines? If not now, what about later?
[9] Will the investment in time and money on a language, its tools, and techniques for this project, pay off by having the language, etc., useful for other projects?

This list is, of course, incomplete, but it gives an idea of what influences the choice of a programming language for a production system.

Notice that the nature of the language itself is directly relevant to only the first, second, and last questions on the list. The other questions pertain mostly to the available implementations and to the community of users that surrounds the language(s) and implementations, the infrastructure of the language. This implies that to be successful, a language must be designed with an eye to its likely implementations, the communities of people who will be using it, and the purposes to which they will put it.

7.2 C++ and C

C++ was originally intended for the same kinds of applications as C. Although C started out as a language in which to write operating systems, it has since been used for a wide variety of things that fall into the general category of ‘‘system programming.’’ Such things often need to get at particular hardware facilities through extralinguistic means. That implies that to remain useful for system programming, C++ must be careful not to stray too far from the underlying machine (see §2).

Because C++ was intended to be useful in the same areas as C, one major goal of C++ has been to do everything C can, and do it as efficiently in time and space as C. Consequently, if one writes a C program in C++, that program will be as fast and small as it would have been in C. This is not true of every C++ implementation, of course, but attainable in theory, and often achieved in practice. C++ even made a few improvements on C in areas not related to abstraction. Some of those improvements, such as `const` types and the ability to include argument types as part of a function declaration, found their way back into C. Others, such as inline function definitions, did not.

The desire to do everything C can do is a strong constraint on C++. For example, it has made it infeasible to make the primitive C++ array and pointer operations any safer than their C counterparts. It is possible, of course, to define safe data structures as C++ classes, but in practice few C++ programmers have the discipline needed to use such data structures exclusively. Thus, C is both a great strength of C++ and a great weakness.

That C++'s relationship with C wouldn't be easy was clear from the start. We like aspects of C, but some key elements of the C language and culture are most disruptive to people trying to write more abstract programs and trying to reason about programs. For example, the C preprocessor is essential for real-world C programming, but is also a menace: any piece of source text may turn out not to be what it appears to be because a macro substitution may radically change what the programmer wrote before the compiler sees it.

The traditional academic response to such problems seems to be ''ban it!'' The C++ answer has been: first make the obnoxious feature redundant, then discourage its use; finally we may actually consider banning the now-unused feature. This strategy is slow and often frustrating, but it respects people's practical needs in a way a more radical approach doesn't. C++ doesn't yet have facilities that make the C preprocessor completely redundant, but inline functions, constants, namespaces, templates, etc., allow a programmer to restrict the use of preprocessor facilities to a minimum related to source code management.

The policy regarding C/C++ compatibility has been expressed as: ''As close to C as possible—but no closer'' [Koenig,1989]. In practice, this means that C++ accepts any C feature—however ugly—as long as it does not interfere with the type system. This policy has kept incompatibilities to an easily manageable minimum.

C is the *de facto* measure of efficiency. People generally accept that if something runs as fast as well-written C it is fast enough. If it doesn't, criticism results—fair or not. Since its inception, one of the aims of C++ has been to make it possible to write programs that are not only abstract, but also run quickly. Throughout the lifetime of C++, and well before it, people have argued that such emphasis on run-time performance is unnecessary.

The typical argument runs something like this: ''Computers are so fast these days that we can afford to give up some of that speed if by doing so we gain something in exchange.'' That something might be development time, or safety, or whatever the favorite language of the person making the argument has to offer. Such arguments are often valid, but not always, and it is not easy to tell when they will be important and when they will not.

For small programs—such as many student projects and prototypes—efficiency rarely matters. Larger systems, however, often consists of many layers of software. If overhead is allowed to build up in the individual layers, the total system becomes glacial. Naturally, if the overhead in an individual layer is really support for subsequent layers so that these layers become simpler and faster then this doesn't happen. Unfortunately, we have found this happy phenomenon less common than one might have hoped. In the absence of such synergies, the language with the most efficient low-level semantics—that is, C or C++—wins. Of course, when one is developing programs for one's self or one's immediate circle, such issues are less important. That is one of the ways in which C++ has been guided by the requirements of commercial, rather than academic, users.

Finally, there are application areas where efficiency is paramount. If you are writing an operating systems kernel or a network driver you don't want any fat on your code—for any reason. For hard real-time applications you have the additional requirement that the performance of every feature must be absolutely predictable as well as sufficiently fast. C++ meets the requirements here.

By being C-compatible, C++ was able to benefit from C's libraries easily, directly, and without overheads. The benefits of that are inestimable because it gives the C++ programmer access to the largest collection of new and old code available. It made the difference between early C++ being a toy and being a tool. In addition to gaining access to libraries written in C, link and layout compatibility with C allows C++

programs to call routines in languages with a C compatible calling sequence, such as Fortran and assembler on many systems. Further, C++ functions can be called from such languages. This allowed C++ to be used to write libraries for use from other languages from day one.

7.3 C++ and Other Languages

We have seen other languages as a fertile source of ideas. Programming languages are fun to play with and it is hard to imagine a modern language—except purely commercial hacks—from which one cannot learn something important.

One cannot simply graft a feature from one language on to another, however. The influence in more subtle. In addition to the ‘‘parent languages’’ C and Simula, we can see traces of Ada, Algol68, Clu, and ML in C++. Many more languages, including Lisp, have inspired programming techniques.

Judging from the net, discussions in the literature, remarks at conferences, etc., the relationship between languages is supposed to be antagonistic and dominated by fierce commercial rivalries. This has left few—if any—traces in the definition of C++, and ‘‘marketing strategy’’ never took significant amount of time from the technical work on C++. The primary reason for this is that in the early years (say until 1989), C++ didn’t have any marketing. It grew as a completely disorganized grass-roots movement. The repeated rumors that C++ succeeded because of ‘‘large corporation backing’’ are the products of overexcited imaginations of would-be commercial competitors. In fact, AT&T spent the grand sum of $3000 on C++ advertising in the critical 1985-1989 period. No one could seriously attribute C++’s success to that. The other theory, that C++ was first in the field of OO languages and thus established itself before any competition arose, is equally at odds with facts. C++ became commercially available in October 1985. Ada, Smalltalk, Objective C, and some Lisp dialects were commercially available well before that and even Eiffel was at most half a year behind.

In explaining C++’s success, we fall back on less interesting reasons: C++ was a reasonable language, cheap, easy to port, fast, coexisted well with other languages, and relatively easy to learn.

C++ owes a good part of its success to the fact that it was able to build on the existing C community. In our experience, the only alternative to building on an existing community is to pick a set of problems for which no widely acceptable solution exists.

The first C++ compiler—and the only one for several years—compiled C++ into C instead of assembly language or machine language. This allowed people to port C++ to a new computer in a matter of days instead of months, provided only that the machine already had a C compiler. This guaranteed that C++ could be readily made available on any machine that supported C, which opened the entire C community as potential C++ users. This approach has since become popular as a method of making new languages available [Stroustrup,1994§3.3].

The programming language one uses will be influenced in practice by the languages one’s neighbors use. This is true of natural languages as well as programming languages, of course. English is the world’s most widespread second language. The reason is that lots of people speak it already, that there is a significant English literature, that more technical information is available in English than in other languages, etc. The facts that English (as actually used) doesn’t have a fixed grammar, that English spelling is an arcane art, that idiomatic English is fiendishly difficult and varies from place to place and from time to time, that English has more words than any other language, etc. don’t seem to matter. The benefits of knowing English make it worth more effort than most other languages. In addition, it is relatively easy to speak English badly, and people accept poor English as long as it effectively conveys information. We believe similar phenomena are occurring with C++.

7.4 Learning to use C++

In two hours, it is possible to teach a C programmer enough C++ to make that programmer noticeably more productive. In a week, it is possible to teach a C or Pascal programmer enough to be a functioning C++ programmer in the sense of being able to write code without looking in the manual all the time. It usually takes six to eighteen months for a programmer to become genuinely comfortable with object-oriented design to the point where the proper use of most C++ language features feels natural enough to be unnoticed in the larger task of building software.

Exceptional programmers can do better yet. However, C++ wasn’t designed for exceptional programmers. You don’t have to be a genius to be a good C++ programmer.

The estimate of the time needed to become comfortable with C++ and object-oriented design is based on the assumption that the programmer/designer learns on the job and stays productive—usually by programming in a ‘‘less adventurous’’ style of C++ during that period. If one could devote full time to learning C++, one would be comfortable faster. However, without application of the new ideas on real projects that degree of comfort could be misleading. Object-oriented programming and object-oriented design are practical—rather then theoretical—disciplines. Unapplied, or applied only to toy examples, these ideas can become dangerous ‘‘religions.’’

The time-consuming thing to learn about C++ is not syntax, but design concepts. A good indication of poor appreciation of C++ is code littered with casts (explicit type conversions). Often, the casts are the result of someone writing C or trying to write Smalltalk in C++.

Our observation is that most people who are aware that there is something to be learned can learn C++ well in a reasonable amount of time. The people who fail, and in consequence write appalling C++ and complain a lot, are in our experience mostly people who approach C++ with the attitude that they know all there is to know about programming so that all they have to do is to pick up ‘‘a bit of odd syntax.’’ Unfortunately, some such people proceed to teach C++ or even write C++ textbooks, and their students then suffer with them.

8 Conclusions

The C++ programming language has evolved in response to its user community. Managing that evolution hasn’t been easy, but new language features, techniques, and libraries had to be developed to meet the needs of a growing user community. The coming ISO/ANSI standard should herald a period of stability of the language definition that ought to set of an explosion of work on tools, techniques, and libraries.

The key problem is education. To use C++ well—or any other language supporting abstraction mechanisms—people must focus on design issues, and teaching design on a large scale is not easy.

9 Acknowledgements

Vince Russo made our Christmas preparations more interesting by suggesting that we might be able to write this paper at the same time. Section 4 was partly inspired by Alex Stepanov’s work on the STL [Stepanov,1994]. Section 5 was partly inspired by [Gamma,1994]. Brian Kernighan made constructive comments of an draft of this paper.

10 References

[Booch,1993] Grady Booch: *Object-oriented Analysis and Design with Applications, 2nd edition*. Benjamin Cummings, Redwood City, CA. 1993. ISBN 0-8053-5340-2.

[Gamma,1994] Gamma, et.al.: *Design Patterns*. Addison Wesley. 1994. ISBN 0-201-63361-2.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *As Close as Possible to C—but no Closer* The C++ Report. Vol 1 No 7 July 1989.

[Koenig,1995] Andrew Koenig (editor): The Working Papers for the ANSI-X3J16 /ISO-SC22-WG21 C++ standards committee.

[Koenig,1995a] Andrew Koenig and Barbara Moo: *Ruminations on C++*. Book, to appear 1996.

[Stroustrup,1985] Bjarne Stroustrup: *The C++ Programming Language*. Addison Wesley, ISBN 0-201-12078-X. October 1985.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (2nd Edition)* Addison Wesley, ISBN 0-201-53992-6. June 1991.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++* Addison Wesley, ISBN 0-201-54330-3. March 1994.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. ISO Programming language C++ project. Doc No: X3J16/94-0095, WG21/N0482. May 1994.

[Vilot,1994] Michael J Vilot: *An Introduction to the STL Library*. The C++ Report. October 1994.