

A Perspective on ISO C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1 Introduction

As C++ programmers, we already feel the impact of the work of the ANSI/ISO C++ standards committee. Yet the ink is hardly dry on the first official *draft* of the standard. Already, we can use language features only hinted at in the ARM [Ellis,1989] and “The C++ Programming Language (second edition),” [Stroustrup,1991], compilers are beginning to show improved compatibility, implementations of the new standard library are appearing, and the recent relative stability of the language definition is allowing extra effort to be spent on implementation quality and tools. This is only the beginning.

We can now with some confidence imagine the post-standard C++ world. To me, it looks good and exciting. I am confident that it will give me something I have been working towards for about sixteen years: a language in which I can express my ideas directly; a language suitable for building large, demanding, efficient, real-world systems; a language supported by a great standard library and effective tools. I am confident because most of the parts of the puzzle are already commercially available and tested in real use. The standard will help us to make all of those parts available to hundreds of thousands or maybe even millions of programmers. Conversely, those programmers provide the community necessary to support further advances in quality, programming and design techniques, tools, libraries, and environments. What been achieved using C++ so far have exceeded my wildest dreams and we must realistically expect that the best is yet to come.

2 The Language

C++ supports a variety of styles. In other words: C++ is a multi-paradigm programming language. The standards process strengthened that aspect of C++ by providing extensions that didn’t just support one narrow view of programming, but made several styles easier and safer to use in C++. Importantly, these advances has not been bought at the expense of run-time efficiency.

At the beginning of the standards process templates were considered experimental; now they are an integral part of the language, more flexible than originally specified, and an essential foundation for standard library. Generic programming based on templates is now a major tool for C++ programmers.

The support for object-oriented programming (programming using class hierarchies) was strengthened by the provision for run-time type identification, the relaxation of the overriding rules, and the ability to forward declare nested classes.

Large-scale programming – in any style – received major new support from the exception and namespace mechanisms. Like templates, exceptions were considered experimental at the beginning of the standards process. Namespaces evolved from the efforts of many people to find a solution to the problems with name clashes and from efforts to find a way to express logical groupings to complement or replace the facilities for physical grouping provided by the extra-linguistic notion of source and header files.

Several minor features were added to make general programming safer and more convenient by allowing the programmer to state more precisely the purpose of some code. The most visible of those are the `bool` type, the explicit type conversion operators, the ability to declare variables in conditions, and the ability to restrict user-defined conversions to explicit construction.

A description of the new features and some of the reasoning that led to their adoption can be found in D&E [Stroustrup,1994]. So can discussions of older features and of features that were considered but

didn't make it into C++.

The new features are the most visible changes to the language. However, the cumulative effect of minute changes to more obscure corners of the language and thousands of clarifications of its specification is greater than the effect of any extension. These improvements are essentially invisible to the programmer writing ordinary production code, but their importance to libraries, portability, and compatibility of implementations cannot be overestimated. The minute changes and clarifications also consumed a large majority of the committee's efforts. That is, I believe, also the way things ought to be.

For good *and* bad, the principle of C++ being "as close to C as possible – and no closer [Koenig,1989]" was repeatedly re-affirmed. C compatibility has been slightly strengthened, and the remaining incompatibilities documented in detail. Basically, if you are a practical programmer rather than a conformance tester, and if you use function prototypes consistently, C appears to be a subset of C++. The fact that every example in K&R2 [Kernighan,1988] is (also) a C++ program is no fluke.

2.1 Coherence

ISO C++ is not just a more powerful language than the C++ presented in "The C++ Programming Language (second edition);" it is also more coherent and a better approximation of my original view of what C++ should be.

The fundamental concept of a statically typed language relying on classes and virtual functions to support object-oriented programming, templates to support generic programming, and providing low-level facilities to support detailed systems programming is sound. I don't think this statement can be proven in any strict sense, but I have seen enough great C++ code and enough successful large-scale projects using C++ for it to satisfy me of its validity.

You can also write ghastly code in C++, but so what? We can do that in any language. In the hands of people who have bothered learning its fairly simple key concepts, C++ is helpful in guiding program organization and in detecting errors.

C++ is not a "kitchen sink language" as evil tongues are fond of claiming. Its features are mutually reinforcing and all have a place in supporting C++'s intended range of design and programming styles. Everyone agrees that C++ could be improved by removing features. However, there is absolutely no agreement which features could be removed. In this, C++ resembles C.

During standardization, only one feature that I don't like was added. We can now initialize a static constant of integral type with a constant expression within a class definition. For example:

```
class X {                                // in .h file
    static const int c = 42;
    char v[c];
    // ...
};

int X::c = 42;                          // in .c file
```

I consider this half-baked and prefer:

```
class X {
    enum { c = 42 };
    char v[c];
    // ...
};
```

I also oppose a generalization of in-class initialization as an undesirable complication for both implementors and users. However, this is an example where reasonable people can agree to disagree. Standardization is a democratic process, and I certainly don't get my way all the time – nor should any person or group.

2.2 An Example

Enough talk! Here is an example that illustrates many of the new language features[†]. It is one answer to

[†] Borrowed with minor changes (and with permission from the author :-)) from D&E [Stroustrup,1994].

the common question “how can I read objects from a stream, determine that they are of acceptable types, and then use them?” For example:

```
void user(istream& ss)
{
    io_obj* p = get_obj(ss); // read object from stream

    if (Shape* sp = dynamic_cast<Shape*>(p)) { // is it a Shape?
        sp->draw(); // use the Shape
        // ...
    }
    else // oops: non-shape in Shape file
        throw unexpected_shape();
}
```

The function `user()` deals with shapes exclusively through the abstract class `Shape` and can therefore use every kind of shape. The construct

```
dynamic_cast<Shape*>(p)
```

performs a run-time check to see whether `p` really points to an object of class `Shape` or a class derived from `Shape`. If so, it returns a pointer to the `Shape` part of that object. If not, it returns the null pointer. Unsurprisingly, this is called a dynamic cast. It is the primary facility provided to allow users to take advantage of run-time type information (RTTI). The `dynamic_cast` allows convenient use of RTTI where necessary without encouraging switching on type fields.

The use of `dynamic_cast` here is essential because the object I/O system can deal with many other kinds of objects and the user may accidentally have opened a file containing perfectly good objects of classes that this user has never heard of.

Note the declaration in the condition of the if-statement:

```
if (Shape* sp = dynamic_cast<Shape*>(p)) { ... }
```

The variable `sp` is declared within the condition, initialized, and its value checked to determine which branch of the if-statement is executed. A variable declared in a condition must be initialized and is in scope in the statements controlled by the condition (only). This is both more concise and less error-prone than separating the declaration, the initialization, or the test from each other and leaving the variable around after the end of its intended use the way it is traditionally done:

```
Shape* sp = dynamic_cast<Shape*>(p);
if (sp) { ... }
// sp in scope here
```

This “miniature object I/O system” assumes that every object read or written is of a class derived from `io_obj`. Class `io_obj` must be a polymorphic type to allow us to use `dynamic_cast`. For example:

```
class io_obj { // polymorphic
public:
    virtual io_obj* clone();
    virtual ~io_obj() { }
};
```

The critical function in the object I/O system is `get_obj()` that reads data from an `istream` and creates class objects based on that data. Let me assume that the data representing an object on an input stream is prefixed by a string identifying the object’s class. The job of `get_obj()` is to read that string prefix and call a function capable of creating and initializing an object of the right class. For example:

```
typedef io_obj* (*PF)(istream&);

map<string,PF> io_map; // maps strings to creation functions

io_obj* get_obj(istream& s)
{
    string str;
    if (get_word(s,str) == false) // read initial word into str
        throw no_class();

    PF f = io_map[str]; // lookup 'str' to get function
    if (f == 0) throw unknown_class(); // no match for 'str'
    return f(s); // construct object from stream
}
```

The map called `io_map` is an associative array that holds pairs of name strings and functions that can construct objects of the class with that name. The associative array is one of the most useful and efficient data structures in any language. This particular map type is taken from the C++ standard library. So is the `string` class.

The `get_obj()` function throws exceptions to signal errors. An exception thrown by `get_obj()` can be caught by a direct or indirect caller like this:

```
try {
    // ...
    io_obj* p = get_obj(cin);
    // ...
}
catch (no_class) {
    cerr << "format error on input";
    // ...
}
catch (unknown_class) {
    cerr << "unknown class on input";
    // ...
}
```

A catch clause is entered if (and only if) an exception of its specified type is thrown by code in or invoked from the try block.

We could, of course, define class `Shape` the usual way by deriving it from `io_obj` as required by `user()`:

```
class Shape : public io_obj {
    // ...
    virtual void draw() = 0;    // pure virtual function
    // ...
};
```

However, it would be more interesting (and also more realistic) to use some previously defined `Shape` class hierarchy unchanged by incorporating it into a hierarchy that adds the information needed by our I/O system:

```
class iocircle : public Circle, public io_obj {
public:
    io_obj* clone() // override io_obj::clone()
    { return new iocircle(*this); }

    iocircle(istream&); // initialize from input stream

    static iocircle* new_circle(istream& s)
    {
        return new iocircle(s);
    }
    // ...
};
```

The `iocircle(istream&)` constructor initializes an object with data from its `istream` argument. The `new_circle` function is the one put into the `io_map` to make the class known to the object I/O system. For example:

```
io_map["iocircle"]=&iocircle::new_circle;
```

Other shapes are constructed in the same way:

```
class iotriangle : public Triangle, public io_obj {
    // ...
};
```

If the provision of the object I/O scaffolding becomes tedious, a template might be used:

```
template<class T>
class io : public T, public io_obj {
public:
    io_obj* clone() { return new io(*this); }

    io(istream&); // initialize from input stream

    static io* new_io(istream& s)
    {
        return new io(s);
    }
};
```

Given this, we could define `iocircle` like this:

```
typedef io<Circle> iocircle;
```

We would still have to define `io<Circle>::io(istream&)` explicitly, though, because it needs to know about the details of `Circle`.

This simple object I/O system may not do everything anyone ever wanted, but it almost fits on a single page, is general and extensible, is potentially efficient, and the key mechanisms have many uses. Undoubtedly, you would have designed and implemented an object I/O system somewhat differently. Please take a few minutes to consider how this general design strategy compares to your favorite scheme, and also think about what it would take to implement this scheme in pre-ISO C++ or some other language.

3 The Library

I have long regretted that I was not initially able to provide C++ with a good enough standard library. In particular, I would have liked to provide a string class and a set of container classes (such as lists, vectors, and maps). However, I did not know how to design containers that were elegant enough, general enough, and efficient enough to serve the varied needs of the C++ community. Also, until I found the time to design the template mechanism, I had no good way of specifying type-safe containers.

Naturally, most programmers want essentially everything useful included in the standard library. That way, they assume, all what they need will be supplied elegantly, efficiently, and free of charge. Unfortunately, that is just a dream. Facilities will be designed by people with different views of how to do things,

limited time, and limited foresight. Also, implementers will find some way of getting paid for their efforts. As the library grows, so does the chances of mistakes, controversy, inefficiencies, and cost.

Consequently, the scope of the standard library had to be restricted to something a relatively small group of volunteers could handle, something that most people would agree to be important, something most people could easily learn to use, something that would be efficient enough for essentially all people to use, and something C++ implementers could ship without exorbitant cost. In addition, the standard library must be a help, rather than a hinderance, to the C++ library industry.

The facilities provided by the standard library can be classified like this:

- [1] Basic run-time language support (for allocation, RTTI, etc.).
- [2] The standard C library (with very minor modifications to minimize violations of the type system).
- [3] Strings and I/O streams (with support for international character sets and localization).
- [4] A framework of containers (such as, `vector`, `list`, `set`, and `map`) and algorithms using containers (such as general traversals, sorts, and merges).
- [5] Support for numeric computation (complex numbers plus vectors with arithmetic operations, BLAS-like and generalized slices, and semantics designed to ease optimization).

This is quite a considerable set of facilities. The description of the standard library takes up about two thirds of the space in the standards document. Outside the standard C library, the standard C++ library consists mainly of templates. There are dozens of template classes and hundreds of template functions.

The main criteria for including a class in the library was that it would somehow be used by almost every C++ programmer (both novices and experts), that it could be provided in a general form that did not add significant overheads compared to a simpler version of the same facility, and that simple uses should be easy to learn. Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

The standard library is described elsewhere [Stepanov,1994] [Vilot,1994] so let me just give a short – but complete – example of its use:

```
#include <string>           // get the string facilities
#include <fstream>          // get the I/O facilities
#include <vector>           // get the vector
#include <algorithms>       // get the operations on containers

int main()
{
    string from, to;        // standard string of char
    cin >> from >> to;     // get source and target file names

    istream_iterator<string> ii
        = ifstream(from.c_str()); // input iterator
    istream_iterator<string> eos;   // input sentinel

    ostream_iterator<string> oo
        = ofstream(to.c_str());   // output iterator

    vector<string> buf(ii,eos);    // standard vector class
                                   // initialized from input

    sort(buf.begin(),buf.end());  // sort the buffer
    unique_copy(buf.begin(),buf.end(),oo); // copy buffer to
                                           // output discarding
                                           // replicated values

    return ii && oo;            // return error state
}
```

As with any other library, parts will look strange at first glance. However, experience shows that most people get used to it fast.

I/O is done using streams. To avoid overflow problems, the standard library `string` class is used.

The standard container `vector` is used for buffering; its constructor reads input; the standard functions `sort()` and `unique_copy()` sort and output the strings.

Iterators are used to make the input and output streams look like containers that you can read from and write to, respectively. The standard library's notion of a container is built on the notion of "something you can get a series of values from or write a series of values to." To read something you need the place to begin reading from and the place to end; to write simply a place to start writing to. The word used for 'place' in this context is *iterator*.

The standard library's notion of containers, iterators, and algorithms is based on work by Alex Stepanov and others [Stepanov,1994].

4 The Standards Process

Initially, I feared that the standardization effort would lead to confusion and instability. People would clamor for all kinds of changes and improvements, and a large committee was unlikely to have a firm and consistent view of what what aspects of programming and design ought to be directly supported by C++ and how. However, I judged the risks worth the potential benefits. By now, I am sure that I underestimated both the risks and the benefits. I am also confident that we have surmounted most of the technical challenges and will cope with the remaining technical and political problems; they are not as daunting as the many already handled by the committee. I expect that we will have a formally approved standard in a year plus minus a few months. Essentially all of that time will be spent finding precise wording for resolutions we already agree on and ferreting out further obscure details to nail down.

The ISO (international) and ANSI (USA national) standards groups for C++ meet three times a year. The first technical meeting was in 1990, and I have attended every meeting so far. Meetings are hard work; dawn to midnight. I find most committee members great people to work with, but I need a week's vacation to recover from a meeting – which of course I never get.

In addition to coming up with technically sound solutions, the committees are chartered to seek consensus. In this context, consensus means a large majority plus an honest attempt to settle remaining differences. There will be "remaining differences" because there are many things that reasonable people can disagree about.

Anyone willing and able to pay the ANSI membership fee and attend two meetings can vote (unless they work for a company who already has a representative). In addition, members of the committee bring in opinions from a wide variety of sources. Importantly, the national delegations from several countries conduct their own additional meetings and bring what they find to the joint ANSI/ISO meetings.

All in all, this is an open and democratic process. The number of people representing organizations with millions of lines of C++ precludes radical changes. For example, a significant increase or decrease in C compatibility would have been politically infeasible. Explaining anything significant to a large diverse group – such as 80 people at a C++ standards meeting – takes time, and once a problem is understood, building a consensus around a particular solution takes even longer. It seems that for a major change, the time from the initial presentation of an idea until its acceptance was usually a minimum of three meetings; that is, a year.

Fortunately, the aim of the standards process isn't to create the perfect programming language. It was hard at times, but the committee consistently decided to respect real-world constraints – including compatibility with C and older variants of C++. The result was a language with most of its warts intact. However, its run-time efficiency, space efficiency, and flexibility were also preserved, and the integration between features were significantly improved. In addition to showing necessary restraint and respect for existing code, the committee showed commendable courage in addressing the needs for extensions and library facilities.

Curiously enough, the most significant aspect of the committee's work may not be the standard itself. The committee provided a forum where issues could be discussed, proposals could be presented, and where implementers could benefit from the experience of people outside their usual circle. Without this, I suspect C++ would have fractured into competing dialects by now.

5 Challenges

Anyone who thinks the major software development problems are solved by simply by using a better programming language is dangerously naive. We need all the help we can get and a programming language is only part of the picture. A better programming language, such as C++, does help, though. However, like any other tool it must be used in a sensible and competent manner without disregarding other important components of the larger software development process.

I hope that the standard will lead to an increased emphasis on design and programming styles that takes advantage of C++. The weaknesses in the compatibility of current compilers encourage people to use only a subset of the language and provides an excuse for people who for various reasons prefer to use styles from other languages that are sub-optimal for C++. I hope to see such native C++ styles supported by significant new libraries.

I expect the standard to lead to significant improvements in the quality of all kinds of tools and environments. A stable language provides a good target for such work and frees energy that so far has been absorbed tracking an evolving language and incompatible implementations.

C++ and its standard library are better than many considered possible and better than many are willing to believe. Now we “just” have to use it and support it better. This is going to be challenging, interesting, productive, profitable, and fun!

6 Acknowledgements

The first draft of this paper was written at 37,000 feet en route home from the Monterey standards meeting. Had the designer of my laptop provided a longer battery life, this paper would have been longer, though presumably also more thoughtful.

The standard is the work of literally hundreds of volunteers. Many have devoted hundreds of hours of their precious time to the effort. I’m especially grateful to the hard-working practical programmers who have done this in their scant spare time and often at their own expense. I’m also grateful to the thousands of people who – through many channels – have made constructive comments on C++ and its emerging standard.

7 References

- | | |
|-------------------|--|
| [Ellis,1989] | Margaret A. Ellis and Bjarne Stroustrup: <i>The Annotated C++ Reference Manual</i> . Addison-Wesley. Reading, Massachusetts. 1990. |
| [Kernighan,1988] | Brian W. Kernighan and Dennis M. Ritchie: <i>The C Programming Language</i> . Prentice-Hall, Englewood Cliffs, New Jersey. 1978. Second edition 1988. |
| [Koenig,1989] | Andrew Koenig and Bjarne Stroustrup: <i>As Close as Possible to C—but no Closer</i> The C++ Report. Vol 1 No 7 July 1989. |
| [Koenig,1995] | Andrew Koenig (editor): <i>The Working Papers for the ANSI-X3J16 /ISO-SC22-WG21 C++ standards committee</i> . |
| [Stroustrup,1991] | Bjarne Stroustrup: <i>The C++ Programming Language (2nd Edition)</i> Addison Wesley, ISBN 0-201-53992-6. June 1991. |
| [Stroustrup,1994] | Bjarne Stroustrup: <i>The Design and Evolution of C++</i> Addison Wesley, ISBN 0-201-54330-3. March 1994. |
| [Stepanov,1994] | Alexander Stepanov and Meng Lee: <i>The Standard Template Library</i> . ISO Programming language C++ project. Doc No: X3J16/94-0095, WG21/N0482. May 1994. |
| [Vilot,1994] | Michael J Vilot: <i>An Introduction to the STL Library</i> . The C++ Report. October 1994. |