A Brief Look at C++

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This note describes some key aspects of what C++ is and of how C++ has developed over the years. The perspective is that of an experienced C++ user looking at C++ as a practical tool. No attempts are made to compare C++ to other languages, though I have tried to answer some questions that I have often heard asked by Lisp programmers.

1 Introduction

Like all living languages, C++ has grown and changed over the years. For me, the improvements to C++ have seemed almost glacially slow and inevitable; they are natural developments of C++'s own internal logic and deliberate responses to the experiences of hundred of thousands of users. To many who wanted to use C++ aggressively, this growth has been frustratingly slow and timid. To some who considered C++ only infrequently, the developments have seem like unpredictable lurches into the unknown. To others, C++ has simply been something peripheral about which little concrete was known, but about which a multitude of strange rumors persisted.

However you look at it, C++ has developed significantly since its first appearance. As an example, consider a simple function that sorts a container and then counts the number of entries between `Dahl` and `Nygaard`:

```
template<class C> int cnt(C& v)
{
        sort(c.begin(),v.end());
        C::iterator d = find(v.begin(), v.end(), "Dahl");
        return count(d, find(d, v.end(), "Nygaard"));
}
```

A container is seen as a sequence of elements from `begin()` to `end()`. An `iterator` identifies an element in a container.

This template function will work as described for any container that conforms to the conventions of the C++ standard library with elements that can be compared to string literals. For example:

```
vector<char*> v;      // vector of C-style strings
list<string> lst;     // list of C++ strings
// ...
int i1 = cnt(v);
int i2 = cnt(lst);
```

The types `vector`, `list`, and `string` are parts of the standard C++ library.

Naturally, we need not build the string values `Dahl` and `Nyggard` into our little function. In fact, it is easy to generalize the function to do perform an arbitrary action on a range of values of arbitrary types in a container of arbitrary type.

Clearly, this style of programming is far from traditional C programming. However, C++ has not lost touch with C's primary virtues: flexibility and efficiency. For example, the C++ standard library algorithm `sort()` used here is for many simple and realistic examples several times faster than the C standard library `qsort()` function.

2 The C++ Standard

C++ is a statically-typed general-purpose 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. That fundamental concept is sound. I don't think this 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.

By 1989, the number of C++ users and the number of independent C++ implementors and tools providers made standardization inevitable. The alternative was to allow C++ to fracture into dialects. In 1995, the ANSI and ISO C++ standards committees reached a level of stability of the language and standard library features and a degree of precision of the description that allowed a draft standard to be issued [Koenig,1995]. A formal standard is likely in late 1996 or early 1997.

During standardization, significant features and libraries were added to C++. In general, the standards process confirmed and strengthened the fundamental nature of C++ and made it more coherent. A description of the new features and some of the reasoning that led to their adoption can be found in [Stroustrup,1994]. So can discussions of older features and of features that were considered but didn't make it into C++.

2.1 Language Features

Basically Standard C++ is the language described in ''The C++ Programming Language (2nd edition)'' [Stroustrup,1991] with namespaces, run-time type information, plus a few minor features added. Among the many minor improvements, the refinements to the template mechanisms are the most significant.

Here is one of the classical examples of object-oriented programming in C++:

```
class Shape {
        virtual void draw() = 0:
        virtual void rotate(int) = 0;
        // ...
};
```

Class `Shape` is an abstract class; that is, a type that specifies an interface, but no implementation. Specific types that conforms to that interface can be defined. For example, this defines `Circle` to be a kind of `Shape`:

```
class Circle : public Shape {
        Point center;
        int radius;
public:
        Circle(Point, int);  // constructor
        void draw();
        void rotate(int) { }
        // ...
};
```

We can now manipulate all kinds of shapes through their common interface. For example, this function rotates a vector of arbitrary `Shape`s `r` degrees:

```
void rotate_all(vector<Shape*>& v, int r)
{
        for (int i = 0; i<v.size(); i++) v[i]->rotate(r);
}
```

For each `Shape`, the appropriate `rotate()` is called. In particular, if the `Shape` rotated is a `Circle`, `Circle::rotate()` is called.

Consider reading `Shape`s from a stream:

```
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();
}
```

Here, the `dynamic_cast` operator is used to check that the objects really are `Shape`s. Any kind of `Shape`, for example a `Circle`, is acceptable. We throw an exception if an object that is not a `Shape` is encountered.

This example is rather trivial. However, the techniques presented and the language features supporting them have been used in the construction of some of the largest and most demanding applications ever built.

2.2 The Standard Library

The lack of a solid standard library has always been one of C++'s greatest weaknesses. This lack caused a proliferation of incompatible ''foundation libraries,'' and diverted novice C++ programmers from getting real work done into designing basic library facilities. The latter was particularly nasty because it is far harder to design and implement good basic library facilities than using them. This lack of a standard library forced many programmers to deal with advanced C++ features before they had mastered the basics.

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`, 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).

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.

Every algorithm works with every container without the use of conversions. This framework, conventionally called the STL [Stepanov,1994], is extensible in the sense that users can easily provide containers and algorithms in addition to the ones provided as part of the standard and have these work directly with the standard containers and algorithms.

3 Tools, Environments, and Libraries

The stability resulting from by the near-completion of the standard is causing a boom in work on programming environments, libraries, and tools. Traditionally, much of the effort in the C++ world has been aimed at producing a language that could be used effectively for significant industrial projects even in the absence of advanced tools and environments. This has not stopped excellent C++ tools and environments from appearing, but the evolution of the language has been a significant drag on the C++ implementation and tools communities.

I expect to see really great program development environments to become almost universal over the next few years. Already, features that people had deemed impossible for C++ are available in shipped products. For example, Sun's C++ implementation allows you to stop an executing program at a breakpoint, rewrite a function, and re-start using the new function. This is something that people using interpreted and dynamically-typed languages have taken for granted for decades. However, it is one interesting and significant step towards my goal of a program development environment combining the strengths of a statically-typed language with the benefits of a sophisticated environment like the ones usually associated with dynamically-typed languages.

One of the benefits of a large user community is the availability of libraries. By now, there is a bewildering variety of C++ libraries, but the development of libraries have been hampered by both the differences between compilers and by the lack of a standard library. The former problem has led to unnecessary segmentation of the community, and to the emergence of libraries especially designed to allow crossplatform development. The latter problem forced library developers to re-invent basic concepts, such as string and list, over and over again. Though it will takes years to work these problems out of the system, we now have an alternative and can get on with more important and interesting tasks.

Automatic garbage collection is possibly the issue over which the C and Lisp communities has traditionally been most at odds. The Lisp community was certain that memory management was far too important to leave to users, and the C community was sure that memory management was far too important to leave to the system. C++ takes an intermediate approach. Automatic garbage collection is possible, but not compulsory in C++. Traditionally, this simply meant that C++ programs didn't use automatic garbage collection, but now both commercial and free garbage collectors for C++ has found their way into non-experimental use. The performance of these collectors is respectable, and in particular, far better than the pessimistic predictions that I have repeatedly heard over the years. Even where a garbage collector isn't used, well-designed C++ programs suffer far less from memory management problems than traditional C programs. Memory management is often encapsulated in user defined types so that users don't have to allocate and deallocate memory directly. In particular, standard containers such as `string`, `vector`, and `list` do their own memory management and provide variable-sized data structures.

4 Programming Styles

C++ is a multi-paradigm language. In other words, C++ was designed to support a range of styles. No single language can support every style. However, a variety of styles that can be supported within the framework of a single language. Where this can be done, significant benefits arise from sharing a common type system, a common toolset, etc. These technical advantages translates into important practical benefits such as enabling groups with moderately differing needs to share a language rather than having to apply a number of specialized languages.

For starters, C++ supports traditional C-style. Other styles emphasize the use of classes, abstract classes, class hierarchies, and templates to express concepts and relationships between concepts directly, cleanly, and affordably. For example, §1 used generic programming and §2.1 demonstrated abstract classes and class hierarchies.

Much of the work on styles (for example [Kornig,1995b]) and patterns (for example, [Gamma,1994]) in the C++ community has focussed on finding ways to express ideas from a variety of languages and systems in a way that can be effectively and efficiently be utilized by C++ programmers writing larger production systems. The emphasis is on effective use of C++'s flexible and extensible static type system.

5 Acknowledgements

Thanks to Craig Knoblock for inviting me to write this note.

6 References

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

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

[Koenig,1995b] Andrew Koenig and Bjarne Stroustrup: *Foundations for Native C++ Styles*. Software – Practice & Experience. 1995.

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

[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.