Directions for C++

Paper Number 4004

Joseph Coha
ANSI C++ Project Manager
Hewlett-Packard Company
11000 Wolfe Rd.
Cupertino, CA 95014

C++ Present and History

Today's C++ compilers each implement a subset of the features described in the ANSI/ISO C++ Draft Working Paper. Some compiler vendors have been conservative in modifying their implementation as the standard has evolved. Others have released new versions trying to closely track the progress made by the ANSI/ISO C++ committee as the language becomes more well-defined. The immediate impact has been porting problems for software developers who are required to sell their products on multiple platforms. Also, C++ software tools often do not support the latest features of the language because the C++ software tool vendors have had a difficult time keeping their products up to date. Software development environments that support the full C++ language including its many new features have been scarce. And, of course, newly introduced features have caused compiler instability.

If we look back over the last several years, we can appreciate the progress that has been made in defining the C++ language. In March, 1990, the *Annotated C++ Reference Manual* (ARM) was adopted as the base document for the standard by the ANSI committee. At the time, remember that much of the C code being written was not even type-safe. Also, the newly introduced 2.0 version of cfront had not yet implemented either templates or exception handling. Since 1990, the C++ language has evolved and improved under the guidance of the ANSI X3J16 and ISO WG21 working groups. The addition of new features has resulted in a language that will better meet the needs of its users. But the new features have also caused the language's instability.

Let's look at the areas of templates, exception handling, runtime type identification, namespaces as well as the resolution of some of the issues involved in generating a consistent C++ language definition. We'll also explore additional benefits resulting from a stable language definition. Here is a brief historical overview of the features that we will cover.

1990 cfront 2.1 - ARM
1991 cfront 3.0 - Templates
1992 cfront 3.0+ - Exception Handling
1993 Runtime type identification; Namespaces

1994 HP Standard Template Library added to Draft Working Paper
1995 Publication of the Committee Draft for public review

Templates

Early C++ users relied on macros to provide a generic class mechanism. The problem with the use of macros is that the substitutions are not type-safe. Anyone who has used this programming style also knows that the resulting code is extremely difficult to debug.

The section on templates in the ARM is described as a placeholder, has only a minimal description of the facility, and is explicitly labeled as experimental. The templates section of the ARM was first incorporated into the standard in 1990. An implementation of templates was included in the 3.0 version of cfront released in 1991.

As compiler implementers and users gained more experience with templates, they brought their questions back to the ANSI/ISO committee. As a result, the committee began logging the issues and their proposed resolutions. Version 14 of the templates issues document was circulated before the ANSI/ISO C++ meeting in March, 1996 [Spicer]. The document lists 113 closed issues and at least another 65 that have been both closed and removed from the list. The resolution of such a large number of complex issues represents a tremendous accomplishment by the committee.

A class template defines a parameterized type which, when supplied with types (or values) for the parameters, generates a specialization of the template that is a type. The member functions of the class template are member function templates that have the same parameters as the class template. For those of you unfamiliar with templates, the following example illustrates their use.

```
template <class T>                  // T is the template parameter
class Holder {                      // Holder is the name of the class template
  public:
    Holder();
    void Hold(const T&);            // Type of the formal parameter not known until type of T
                                    //    is known
    [ ... ]                         // Remainder of class template declaration
};
```

The Hold() template function definition can be either within or following the class template declaration.

```
template <class T>                  // T is the template parameter
void Holder<T>::Hold(const T& t) {
    Order();
    Check(t);                       // Check implicitly depends on the template argument
    [ ... ]                         // Remainder of function template definition
}
```

The template is instantiated by the compiler on the template's first use in the source code being compiled. The instantiation is done implicitly by the compiler, creating a compiler-generated specialization. But where exactly does the compiler create the specialization? The point is specified in the Working Paper as the nearest enclosing namespace scope surrounding the use. Thus, names in that scope are visible to be bound to names in the template. If there is a use of Holder<int>::Hold(const int&) the compiler looks up which function should be called to satisfy the call to Check(int) in the nearest enclosing namespace. In the following example, the call is made to Check(int).

```
// include declaration of template <class T> Holder
// include definition of constructor and template <class T> void Holder<T>::Hold(const T& t)
void Check(int i);                  // 2. Function called when Holder<int>::Hold(const int&)
                                    //    specialization created by the compiler
void Check(long l);

void TestFunc()
{
    Holder<int> holder1;
    int i = 5;
    holder1.Hold(i);                // 1. Use creates compiler-generated specialization in
}                                   //    nearest enclosing namespace
```

A question arises. If the first use of the template is in a function or class then does that scope include the name of that function or class? The committee decided that the name of the function or class should be visible. For functions, this makes sense because doing otherwise would prevent recursion. However, the committee also decided that members of the class and local members of a function are not visible.

Another question concerns the point in the program at which names are bound. When the body of a function template contains a call to a function (as is the case in the definition of Hold above), what are the rules for looking up the name of the function to which to bind? And at what point does the name lookup occur? Does the lookup occur at the location where the function template is defined? Or does it occur at the location where the template function is used (the point of instantiation)? The committee considered proposals which would restrict name lookup to one context or the other. Another proposal considered whether finding a name in both contexts should be considered an error.

After much debate, the committee agreed that the presence of a template parameter in the function name (either as a scope qualifier or associated with a parameter) determines the method for function name lookup. In our example, the call to the Check() function has an actual parameter that has a type that is dependent on the type of the template parameter. Check() is said to implicitly depend on the template argument. The dependency means that the Check() function to be called is not known until the point at which the compiler creates the specialization. In our previous example, note that the declaration for void Check(int i) follows the definition of

template <class T> void Holder<T>::Hold(const T& t), but precedes the use of the function Holder<int>::Hold(const int& t).

In our example, both the names of functions in the namespace containing the function template definition and the namespace at the point of instantiation where the compiler creates the specialization are used to resolve the call to Check(). Standard overload resolution is used to select the appropriate Check() to call.

On the other hand, the Order() function is not a dependent name, so the name lookup occurs at the point of the function template definition.

The basis for choosing this model was the recognition by members of the committee that a template writer is generally interested in knowing that calls made to functions that are not dependent on the template arguments are best bound at the point where the template definition occurs. For dependent names, a function call that has either a name scope qualifier that is a template parameter or a formal parameter that depends in some way on a template parameter can be resolved at the use of the template function. This makes sense because at the point in the program's source where the template is used and the specialization created, the declaration of the template parameter has been seen.

The careful thought and consideration which the committee has used in deciding this and the many other template issues helps to make C++ a practical, well-designed language useful for the many purposes required by software development engineers. The committee is fulfilling the early requirements stated in the ARM (page 3) to create a fast, flexible and portable language. As features have been added to the language, the interactions between the new features and those already adopted has raised concerns about consistency. Many of the issues have been resolved. However, the interactions between namespaces and templates will still require additional work by the committee.

Exception Handling and Runtime Type Identification

The exception handling proposal in the ARM outlined a model that provided for throwing objects and catching of types. Thus, a catch clause specifying a single base type, B, can catch objects of that type, B, as well as other objects that have the B class as a base class. In the following example, the throw of the D type object in the called function, Func2(), will be caught by the catch clause in MyFunc().

```
class B              { public: virtual void cause(); };
void B::cause()      { cout << "base" << endl; }

class D: public B    { public:  virtual void cause(); };
void D::cause()      { cout << "derived" << endl; }

void Func2()         { throw D(); }                  // 3.  Func2 throws a D object

void MyFunc() {                                     // 1.  Call is made to MyFunc()
     try { Func2(); }                               // 2.  MyFunc() calls Func2()
     catch ( B& b ) { b.cause(); }                  // 4.  Catch clause of B type catches D object
}
```

Clearly, the C++ run time exception handling mechanism needs to know the type of the object being thrown in order to match it with the types specified in the catch clauses. In the process of implementing exception handling in cfront [Cameron, 1992], the engineers at HP proposed that the type information required to make exception handling work correctly could be added as a language feature [Lenkov, 1991].

Runtime type identification (RTTI) was officially added to the language specification in 1993. Generally, when streaming objects into and out of processes, use of RTTI is essential. Other uses of RTTI are to extend libraries that might not otherwise be extendable, to simplify the overall design of the program or to make the design more efficient [Stroustrup, 1992]. For example, instead of adding a virtual function to a base class, add the function to a derived class and then check in the implementation whether it is appropriate to call that specific function. Although RTTI can be applied to some problems, it should not be used as a substitute for virtual functions when designing your class hierarchy.

RTTI adds two operators, typeid() and dynamic_cast(), and three classes, type_info, bad_cast and bad_typeid to the language definition. The parameter to typeid() can be a polymorphic type, non-polymorphic type or an expression. The typeid() operator returns a const type_info object by reference. Comparison operators, operator== and operator!=, allow the programmer to compare type_info objects. The type_info name() method can be used to return a string that represents the name of the type. Here is the definition of type_info:

```
namespace std {
    class type_info {
    public:
        virtual ~type_info();
        bool operator==(const type_info& rhs) const;
        bool operator!=(const type_info& rhs) const;
        bool before(const type_info& rhs) const;
        const char* name() const;
    private:
        type_info(const type_info& rhs);
        bool operator=(const type_info& rhs);
    };
}
```

The dynamic_cast() operator is particularly useful for correctly casting an object from a base type to a derived type. Object request brokers often use dynamic_cast() in this way to safely downcast (narrow) objects.

```
class B {                                   // Declaration of class B
public:
     virtual void whatami();
 };
void B::whatami() { ... };
class D : public virtual B {                // Declaration of class D
public:
     virtual void whatami();
     void isD();
};
void D::whatami() { ... }
void D::isD() { ... }
void func(B& b) {
     D& d = dynamic_cast<D&>(b);    // 3.  Cast to a type D object
     d.whatami();                   // 4.  Virtual function call
     d.isD();                       // 5.  Non-virtual call requires that the object be of type D
}
int main() {
     D d;                           // 1.  Create an object of type D
     func(d);                       // 2.  Cast to a B reference for the call
[ ... ]
```

In our simple example, the cast will succeed. In a real program, what would happen if the dynamic_cast() fails for the object provided? If the example were to have used pointers instead of references, the specified behavior of dynamic_cast() on a bad cast of a B* type to a D* type is to return null. When using pointers, the return value can easily be checked to see if it is null which in turn indicates whether the cast was successful. In our example, since the return value is a reference and references are required to be initialized to refer to a valid object, how do we determine whether the cast was successful? The committee agreed that this was a good use of an exception class, bad_cast. If the cast in our example fails then an object of type bad_cast is thrown. Of course, this requires that we rewrite the definition of func().

```
void func(B& b) {
     try {
          D& d = dynamic_cast<D&>(b);    // Cast to a type D object
     }
     catch ( bad_cast ) {
          [ ... ]                        // Do something
     }
     d.whatami();                        // Virtual function call
     d.isD();                            // Non-virtual call requires that the object be of type D
}
```

Issues with regard to RTTI were still open following the ANSI/ISO C++ meeting in November, 1995. The process of creating the standard is a slow, iterative process, but, as we have seen, the result of the process is a well-defined language specification.

Namespaces

Another latecomer to the standardization process is namespaces, adopted by the committee in 1993. As C++ became more widely used in the late 1980's and 1990's, people developed libraries of both commonly used and domain specific sets of classes. Problems quickly developed with regard to name collisions in the libraries. The most obvious example was the definition of a String class in many of these libraries. Library vendors sometimes worked around this problem by creating a "namespace." In certain cases, the namespace was simply a special prefix for the name of each of the classes provided. Another clever method took advantage of the nested class feature in C++ to wrap the library's classes in an enclosing class. Thus, all of the classes are in the same nested scope and removed from the global name space. However, this solution was impractical because all of the declarations were required to be in a single header file.

Namespaces solve the problem just mentioned by permitting the scope to span multiple, disjoint namespaces that have the same name. The result is that the declarations of a single namespace can be split across files as well. Here is an example of the use of namespaces.

```
namespace N1 {
    int ni;                     // Refer to this data member as N1::ni
    void func(int);             // Refer to this function as N1::func
}
namespace N2 {
    int ni;                     // N2::ni      No conflict with N1::ni
    void func(int);             // N2::func    No conflict with N1::func(int)
}
namespace N1 {                  // Extends namespace N1
    int nni;
}
```

However, because member (class, function and data) names now exist in a scope, each reference to a name requires qualification. The committee addressed the problem of verboseness caused by the requirement for additional qualification by adding using directives and namespace aliasing. Namespace aliasing is a useful method for creating shorter, more manageable, names for deeply nested qualified names or for those namespaces with very long names.

```
namespace abrev = AVeryLongNameOfANamespaceThatSomeoneHasMade
```

A using directive simply makes the names of members in a namespace available in the current scope after the use of the using directive. The names are available as though they had been declared in the nearest enclosing namespace in which the using directive appears. This leads to a number of interesting issues with regard to name lookup. How is the name resolution performed when you have multiple using directives? One of the solutions to this problem that was considered by the committee had strict rules for ordering the search of the namespaces. The issue occurs because each namespace can, in turn, contain additional using directives. If the using directives are looked at in their

lexical order in the source code, how does one proceed, depth first or breadth first? Consider the following example.

```
namespace NM1 {
    int nmi;
}
namespace NM2 {
    int nmi;
    using namespace NM1;          // Using directive makes the names in NM1 visible
}
void set_nmi() {
    using namespace NM2;          // Using directive makes the names in NM2 visible
    nmi=0;                        // Reference to the name nmi is ambiguous because both
                                  //     are visible
}
```

The committee agreed on a name lookup mechanism that is transitive. At the point of the lookup, a merged context is created using all of the names from all of the namespaces specified by the using directives plus all of the names found in any namespace encountered during the processing of any using directive in one of the namespaces being processed. Thus, in our example, both NM1::nmi and NM2::nmi are visible as simply nmi.

From within a namespace, referencing a global function now becomes problematic as well. To address this problem, the unary :: is defined to mean that the name lookup should be in the global scope rather than in any of the namespace scopes.

In order to alias namespace members in your source, using declarations have been defined that add a namespace name to the local scope in which the using declaration occurs. The names are available as though they had been declared in the namespace in which the using declaration appears. As you would expect, it is an error when the name and type of a local function declaration is identical to the name and type of a declaration introduced by a using directive. Here is an example of the use of a using declaration.

```
namespace UD1 {
    void func();
}
namespace UD2 {
    using UD1::func;              // Using declaration - alias for UD1::func()
}
void test() {
    UD2::func();                  // Calls UD1::func();
}
```

Binary stability

Introducing namespaces, RTTI, and exception handling has had an impact on the runtime used by C++ programs. As was mentioned earlier, some C++ compiler

implementers have aggressively followed the standard, always trying to keep their compiler as up to date as possible with the latest approved features. Naturally, this has been difficult for their customers because they are required to repeatedly roll their code to the latest version of the compiler. Releases are difficult to support because a simple patch to the software application cannot be provided. Instead, the whole product needs to be re-released. Meanwhile, the compiler writer spends all of his or her engineering effort updating and fixing the compiler. No time remains for creating the tools necessary to manage C++ source code or make the code more understandable by providing tools which allow the programmer to generate visual abstractions of their source.

Some tool vendors have tried to develop comprehensive C++ software development environments, but none has succeeded. Even building C++ class libraries that encapsulate all of the common system interfaces has not insured success. As C++ has evolved, the systems being built with the language have become more sophisticated and complex. At the same time, the addition of new features destabilizes the compiler and, as mentioned, breaks binary compatibility. Not only the library vendors, but also the tool vendors are subjected to costly rolls to modify their tools to support the newly-added language features. The impact of the changes has been to increase the cost of the software tools that make it easy to program in C++.

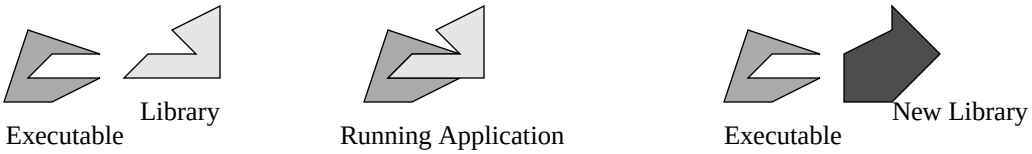
Approximately two years ago, a number of companies made an effort to try to address this issue by specifying an application binary interface (ABI) [Ball]. The reasoning behind their effort was that if the ABI were stable then the tools and library vendors would have a stable environment in which to develop software. Unfortunately, as we have seen, the language changed and, in turn, broke the ABI.

Now that the committee is moving the language in the direction of a stable standard, establishing an ABI once again becomes a possibility. With the increased stability, the C++ compiler implementers will be able to spend less time working on the compiler and more time working on tools to support C++ software development. Already, we are exploring development of new debugging paradigms designed to support object-oriented programming with C++ rather than simply relying on the traditional procedural programming model available with most debuggers.

Release to Release Binary Compatibility

Once we have a stable ABI, we can also start working on the problem that occurs when a base class is changed in a class hierarchy. Generally, making such a change forces a recompile of a large part of a C++ application. If such a change is made to a class which is compiled into an object file that is added to a shared library then any executable that previously was compiled using that shared library will need to be recompiled. If a mechanism could be provided which allowed the executable compiled with the previous version of the shared library to continue to run, then patches to fix problems in large applications written in C++ would be feasible. Also, the same

mechanism could be used in everyday software development to isolate changes made during software development in different areas of large systems. The following diagram illustrates the problem encountered when trying to maintain binary compatibility.



The ability to allow an older version of the executable to run with a new version of a library that contains classes which have been modified is referred to as release to release binary compatibility (RRBC). Of course, without a stable ABI, release to release binary compatibility has not been an addressable problem in C++ applications.

Mechanisms have been developed to provide RRBC, but the costs of doing so have usually been high because of the additional run-time support required. A penalty is generally paid because providing RRBC requires deferring some of the decisions about object layout and the construction of virtual tables from compile time to load time. Generally, the solution also requires additional indirection for access and the associated additional runtime overhead. Creating larger and slower programs is not an encouraging direction, but it may be worthwhile for those applications that require RRBC. Other solutions exist, and decreasing the amount of work required for support of the C++ language should allow more effort to be spent investigating and developing these solutions.

Summary

The C++ language is in its final stages of ratification as an ANSI/ISO standard. The language has benefited greatly from its careful evolution, shepherded by the ANSI/ ISO C++ committee. Useful features have been added that address the needs of the software development community. We have looked at some of these language features, templates, exception handling, runtime type identification and namespaces. With the coming standardization of C++, a new period is opening for software developers which will allow them to shift from compiling and re-compiling code to one in which stable, efficient compilers will become integral tools in C++ software development environments.

Bibliography

Accredited Standard Committee, X3, Information Processing Systems, Working Paper for Draft Proposed International Standard for Information Systems - Programming

Language C++, Document X3J16/96-0018, American National Standards Institute, Washington, D.C., 1996.

Ball, Michael, “The Joys of Stability,” C++ Report, volume 7, number 8, SIGS Publications, New York, October, 1995.

Cameron, Don, Paul Faust, Dmitry Lenkov and Michey Mehta, “A Portable Implementation of C++ Exception Handling,” USENIX C++ Technical Conference Proceedings, Berkeley, CA, 1992.

Ellis, Margaret A. and Bjarne Stroustrup, *The C++ Annotated Reference Manual*, Addison-Wesley Publishing Co., New York, 1990.

Lenkov, Michey Mehta and Shankar Unni, “Type Identification in C++,” USENIX C++ Technical Conference Proceedings, Berkeley, CA, 1991.

Spicer, John, “Template Issues and Proposed Resolutions,” Document X3J16/96-0023, American National Standards Institute, Washington, D.C., 1996.

Stroustrup, Bjarne, *The Design and Evolution of C++*, Addison-Wesley Publishing Co., New York, 1994.

Stroustrup, Bjarne and Dmitry Lenkov, “Run Time Type Identification for C++,” USENIX C++ Technical Conference Proceedings, Berkeley, CA, 1992.