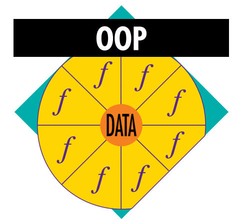
Major Features Adopted by the C++ Standard Committee



By Josee Lajoie

This article provides an overview of the new Runtime Type Information (RTTI) and namespace features and the C++ standard library, as well as a look at the current status of the C++ standard.

After five years of effort, the ANSI/ISO committee responsible for standardizing the C++ programming language is about to complete its task. Five years may seem like a very long time—but the C++ standard committee has accomplished an immense amount of work.

The committee decided to base the C++ Standard on two documents: *The C++ Annotated Reference Manual* (ARM)[1] and the *ISO Standard for the C Programming Language*[2]. It distributed tasks among the following groups:

Core Language Working Group: Led at first by Andrew Koenig and later by me, this group was responsible for clarifying the C++ language described in the ARM and ensuring that the semantics enabled efficient C++ implementations. The group made substantial improvements to many areas of the language, including the name lookup rules, the rules for type conversions and function overload resolution, the memory model, and the object model.

Extensions Working Group: Led by Bjarne Stroustrup, this group was responsible for addressing problems frequently encountered by the C++ programming community. This group only considered problems that could not be solved with existing language mechanisms or for which existing mechanisms were so cumbersome as to make C++ an unacceptable programming solution. This group's work resulted in four major extensions to the base language: templates, exceptions, Runtime Type Information (RTTI), and namespaces.

Library Working Group: Led by Mike Vilot, this group was responsible for providing class libraries that could be used as building blocks for more ambitious class libraries or for other C++ applications.

This article will provide an overview of the major features adopted by the committee. Because templates and exceptions were introduced by the ARM, only an overview of the new RTTI and namespace language features plus an introduction to the C++ standards library are discussed in this article.

> **The ANSI/ISO committee responsible for standardizing the C++ programming language is about to complete its task.**

Runtime Type Information

The basic RTTI support provided in C++ can be broken down into two new facilities: the `dynamic_cast` operator, which allows type conversions that are checked at runtime, and the `typeid` operator, which returns the runtime type of an object.

The dynamic_cast Operator

The syntax for this new operator is `dynamic_cast<T*>(p)`, where `T` represents a type and `p` represents a pointer. The operator converts its operand `p` to the desired type `T*` only if `*p` is really a `T`—that is, only if the runtime type of `*p` is really a `T`. Otherwise, the value of the expression `dynamic_cast<T*>(p)` is 0.

Thus, the `dynamic_cast` operator performs two operations at once. It verifies that the requested cast is valid, and only if it is valid does it perform the cast. This verification renders the `dynamic_cast` much safer than the static (that is,

```
class employee {
public:
  virtual int salary();
};
class manager : public employee {
public:
  int salary();
  virtual int bonus();
};
class programmer : public employee {
public:
  int salary();
};
```

Figure 1. Class hierarchy of polymorphic classes

```
void payroll::calc (employee *pe)
{
   manager *pm =
     dynamic_cast<manager*>(pe);

   // employee salary calculation
   if (pm) {
      // use of manager::bonus()
   }
   else {
      // use of employee's member
         functions
   }
}
```

Figure 2. The dynamic_cast operator

```
class employee {
public:
  virtual int salary();
  virtual int bonus();
};
class manager : public employee {
public:
  int salary();
  int bonus();
};
class programmer : public employee {
public:
  int salary();
  int bonus();
};

void payroll::calc (employee *pe)
{
   // use of pe->salary() and pe->bonus()
}
```

Figure 3. The dynamic_cast replaced by a virtual function call

not runtime-checked) cast used in some situations to cast a base class pointer to a derived class pointer.

The `dynamic_cast` operator can be useful in situations where virtual functions cannot be used. It enables users to safely convert a base class pointer to a derived class pointer, ensuring that the interface of the derived class is only used when it is appropriate. Figure 1 shows a class hierarchy of polymorphic classes.

With a pointer to the base class `employee`, a user can now use the `dynamic_cast` operator to obtain a pointer to the derived class `manager` to use its member function `bonus`. With the `dynamic_cast`, the cast to the derived class pointer is only performed if the cast is valid—that is, if the pointer actually points to a `manager` object. Figure 2 shows what a user of the class hierarchy described above could do.

In the example in Figure 2, if at runtime `pe` actually points to a `manager` object, the `dynamic_cast` operation is successful, and `pm` is initialized to point to a `manager` object. Otherwise, the result of the `dynamic_cast` operation is 0 and `pm` is initialized with the value 0. By checking the value of `pm`, the function `payroll::calc` ensures that the appropriate action is taken for `manager` objects. That is, the `manager::bonus` member function calculates the salary of a `manager` object, while the more general case available calculates the salary of other types of `employee` objects.

Note that the virtual function mechanism already present in C++ is a better choice than RTTI to solve the problem presented. To use virtual functions, however, a user must have access to the source code for the class definitions. To extend the classes with additional virtual functions, a user must modify the base class `employee` and its derived classes to provide new virtual functions (in this particular case, a new `bonus` member function) and provide the necessary overriding definitions for the functions in the derived classes. See Figure 3.

The use of the virtual function mechanism as shown in Figure 3 is much more elegant than the use of the `dynamic_cast` in the earlier example. With the virtual function, the mechanics for the selection of the appropriate virtual function in the derived class are hidden from the users, and the code for the function `payroll::calc` is much cleaner. This mechanism should be preferred over RTTI whenever possible.

RTTI becomes necessary when the user extending the library does not have access to the

source code for the base class or some of the derived classes. For example, the base class could be part of a vendor library, and the source code for this library may not be available to library users. A user who wants to add functionality to the library classes through derivation will be unable to add new virtual member functions to the base classes and derived classes provided in the library. In this case, the user will need to take advantage of RTTI as shown earlier.

The typeid Operator

In addition to the `dynamic_cast` operator, a `typeid` operator is also provided as part of the new RTTI features. The syntax for this operator is as follows:

```
typeid(type_name)    or
typeid(expression)
```

The result of a `typeid` operation has the following type:

```
const type_info &
```

where `type_info` is a class provided in the C++ Standard library that describes the runtime type information provided by the implementation.

The operand of the `typeid` operator can be a type name, in which case `typeid` returns a reference to a `type_info` that represents the type name. The operand of `typeid` can be an expression, in which case `typeid` returns a reference to a `type_info` that represents the type denoted by the expression.

When the operand of `typeid` is a polymorphic type (a class type with virtual functions), the actual object is examined and its dynamic type is returned by the `typeid` operator. When the operand of `typeid` is not a polymorphic type, the `typeid` operator returns the operand's static type.

Suppose there is a need to use the previously described classes `employee` and `manager` with the `typeid` operator. Figure 4 shows how the operator could be used.

These examples show how the `typeid` operator can be used in expressions to compare the runtime type of C++ objects without directly manipulating `type_info` objects.

Let's examine the results of the comparisons provided in the previous example. Each expression compares the result of using the `typeid` operator with an expression operand with the result of using a `typeid` with a type name operand.

```
employee *pe = new manager;
   typeid(pe) == typeid(employee*)    // 1:   True
   typeid(pe) == typeid(manager*)     // 2:   False
   typeid(pe) == typeid(employee)     // 3:   False
   typeid(pe) == typeid(manager)      // 4:   False

   typeid(*pe) == typeid(manager)     // 5:   True
   typeid(*pe) == typeid(employee)    // 6:   False
```

Figure 4. Uses of the typeid operator

First, in `typeid(pe) == typeid(employee*)`, because `pe` is a pointer (not a polymorphic type), the type returned by the expression `typeid(pe)` represents `pe`'s static type, or pointer-to-employee. Note that the expression (`pe`) is examined, not the object type. Therefore, the conditions presented on lines 2, 3, and 4 are false.

This may seem surprising to users accustomed to writing the following:

```
// call to a virtual function
    pe->salary();
```

This results in calling the `salary` member function of the `manager` derived class. In that sense, `typeid(pe)` behaves differently from the virtual function call mechanism. Even if `pe` points to an object of polymorphic type, `typeid(pe)` represents `pe`'s static type, `employee*`.

When the expression '`*pe`' is used with `typeid`, the polymorphism of the object pointed to by `pe` is taken into account. In `typeid(*pe) == typeid(manager)`, because `*pe` is an expression that represents a polymorphic type, the type returned by `typeid` represents the dynamic type of the expression, or `manager`.

As mentioned above, `typeid` can be used with operands of non-polymorphic type. This implies that built-in type expressions as well as constants can be used as operands for the `typeid` operator:

```
int I;
... typeid(I) == typeid(int) ... // True
... typeid(8) == typeid(int) ... // True
```

To use the `typeid` operator, you must include the C++ standard header file, `<typeinfo.h>`. This header file defines the class `type_info`, which describes the runtime type information available and is used to define the type returned by the `typeid` operator. The exact definition of the class `type_info` is implementation defined, but certain features of this class are guaranteed by the

```
class type_info {
    // Implementation dependent representation
  private:
    type_info(const type_info&);                //1
    type_info& operator= (const type_info&);    //2
  public:
    virtual ~type.info();                       //3

    int operator==(const type_info&) const;     //4
    int operator!=(const type_info&) const;     //5

    const char * name() const;                  //6

    int before(const type_info&);               //7
};
```

Figure 5. The type_info class

Vendor A

```
vendor_a.h:
    class complex {  /* ... */  };
    complex sqrt (complex);
    double sqrt (double);
    int sqrt (int);
```

Vendor B

```
vendor_b.h:
    class complex {  /* ... */  };
    complex operator+ (complex c1, complex c2)
       { /* ... */ }
    complex sqrt (complex);
    const double pi = 3.1416;
```

Figure 6. Global names from different libraries

standard. The class is a polymorphic type, which supplies comparison operators and provides a member function that returns the name of the type represented.

Figure 5 represents the interface imposed by the standard for the class `type_info`.

Because the copy constructor and the assignment operator are declared private (lines //1 and //2), users cannot copy `type_info` objects. Because the destructor is declared to be virtual (line //3), the class `type_info` is a polymorphic class type. Lines //4 and //5 declare the overloaded comparison operators. These functions allow two `type_info` objects to be compared, and hence, allow the results obtained by using the `typeid` operator to be compared. The `name` function declared on line //6 returns the name of the type represented by the `type_info` object. C++ users have highly varied requirements for the run-time type information they want an implementation to provide. Some users want to minimize the space overhead resulting from the addition of RTTI to the language. For these reasons, the type name is the only information that the standard mandates that implementations provide.

Namespaces

What problem do namespaces solve? Like the C language, C++ has a single global namespace in which all the names declared in global declarations are entered. This is a difficult situation for library providers and their users. Global names in a library may collide with the global names in a user application, or with global names in other libraries provided by other vendors. For example, vendor A may provide a header file containing certain declarations and definitions, while vendor B may provide a similar header file, both shown in Figure 6.

Given these two header files, a user cannot easily write an application that uses both of these libraries.

Workarounds do exist that allow users to access these two libraries in one application. For example, Figure 7 shows that the problem can be solved when a prefix containing special characters is added to the names in the libraries.

This solution is not very elegant. Using prefixed names in an application can be quite cumbersome, especially if the prefixes are long. Namespaces are a better choice. They do not completely eliminate the global namespace pollution problem, but they significantly reduce the impact of the problem.

Namespace Definitions

A *namespace* is a mechanism for defining a scope. A namespace (user-defined scope) contains the traditional global C++ declarations. A namespace must have a unique global name; it is an error if any other global entity has the same name as a namespace.

A namespace can be used to encapsulate the declarations shown in the previous example:

```
vendor_a.h:
   namespace lib_a {
      class complex {  /* ... */  };
      complex sqrt (complex);
      double sqrt (double);
      int sqrt (int);
   }
```

The names declared within the namespace's braces belong to the `lib_a` namespace and do not collide with global names or names in any other namespaces.

In a namespace, you can declare or define anything you can declare or define at global scope: classes, typedefs, global variables (with their initialization), global functions (with their definition), and templates. The meaning of these declarations and definitions, once wrapped into the namespace, is the same as if the declarations and definitions appeared at global scope. The only difference is that the names declared are different. Namespace members must be referred to by the traditional notation for class members:

```
namespace_name::member_name
```

The members of the `lib_a` namespace can therefore be used as follows:

```
lib_a::complex func(lib_a::complex c) {
  return  ... lib_a::sqrt() ... ;
}
```

It can be cumbersome to always refer to a global name using the notation `namespace_name::member_name`. For this reason, the namespace features offer mechanisms to facilitate the use of namespace members.

Using Declarations

Using declarations enable users to make certain members of a namespace visible without requiring the names of these members to be qualified. For example, the code example above can be rewritten as follows:

```
using lib_a::complex;       //1

complex func(complex c) {
  using lib_a::sqrt;        //2
  return  ... sqrt() ... ;
}
```

Line //1 indicates that from now on, the name `complex` will refer to `lib_a::complex`. Line //2 indicates that, from now on in the body of function `func`, using the name `sqrt` will refer to `lib_a::sqrt`. Line //1 and line //2 are using declarations.

Using Directives

A *using directive* makes the names from the namespace available as if they had been declared at global scope where the namespace was defined. It does not declare local aliases for the namespace member names. For example, the following:

```
namespace A {
   int I, j;
}
```

looks like `int I, j;` to code for which a: `using namespace A;` is in scope. For example:

```
namespace A {
  int I, j, k;
}
void f() {
   using namespace A;
   I = 0;  //1: A::I
   int I;  //2: local declaration of I
}
```

With the using directive `using namespace A;` in function `f`, the reference to `I` on line //1 refers to `A::I`. A using directive is not a declaration; the using directive `using namespace A;` causes the name used on line //1 to appear as if it had been declared outside of `namespace A` in global scope, where `namespace A` was defined. Because of this, a local declaration of the variable `I` is allowed (line //2). It is not a redeclaration of the name `I` in function `f` since no variable `I` is declared in `f` by use of the using directive.

```
vendor_a.h:
   class a_complex {  /* ... */  };
   a_complex a_sqrt (a_complex);
   double a_sqrt (double);
   int a_sqrt (int);


vendor_b.h:
   class b_complex {  /* ... */  };
   b_complex operator+ (b_complex c1, b_complex c2)
      { /* ... */ }
   b_complex b_sqrt (b_complex);
   const double b_pi = 3.1416;
```

Figure 7. Workaround to global name collisions

C++ Standard Library Components

An overview of the C++ standard library will show the following components:

Language support: Describes the types and functions needed to support the C++ language

```
class exception {
 public:
   exception(const exception e);
   virtual ~exception();
   virtual const char * what() const;
   // const char * describes nature of
exception thrown
 };
```

Figure 8. The exception class

features. This portion of the library includes the functions needed for dynamic memory management (`new`, `delete`) and exception processing (`unexpected`, `terminate`), as well as the class needed for RTTI (class `type_info`).

The technical work of the committee should be completed by the end of 1995 and the C++ Standard published by the end of 1996.

Predefined exceptions: Describes classes that support uniform error reporting from library components. These exceptions can be used in any kind of C++ application.

A common base class called `exception` is provided. This class defines a `what` function that can be used to identify the kind of error encountered, as shown in Figure 8.

Two classes are derived from this `exception` class. These two classes describe the two categories of exceptions that can be identified by library components. The class `logic_error` is used for exceptions due to the internal logic of the program (for example, a `logic_error` occurs when a string is accessed past the end of the string), while the class `runtime_error` is used to signal events beyond the scope of the program.

Strings library: Describes components that manipulate sequences of `characters`, where a character can be a `char`, a `wchar_t`, or any other C++ type that has a character-like behavior. This library defines the `basic_string` template, which gives a string the following properties:

- First element of the string is located at position 0
- Number of elements is fixed at construction time
- Elements are stored in contiguous storage

Localization library: Describes components that perform localization of strings. It provides internationalization support for character classification and collation, numeric representation, currency punctuation, and date and time formatting.

Containers library: Describes components that organize collections of objects. It provides template classes that describe the container types (list, vector, stack, queue, and so on).

Iterators library: Describes components that are used to iterate over containers, streams, and stream buffers. The access methods provided by this library vary in functionality and performance. The access methods provide functionality such as insert, reverse, and iterate.

Algorithms library: Describes components that perform algorithmic operations on containers and other sequences. Operations include the following:

- Non-mutating operations (find, count, search)
- Mutating operations (copy, swap, replace, fill, remove reverse, rotate)
- Sorting (binary search, sort)
- Merge
- Heap (push, pop)

Numerics library: Extends the C++ support for numeric processing. It provides a template class to define complex numbers, as well as other types and algorithms heavily used in numerical analysis applications. In general, this library is provided for users who want to use C++ for programming á la FORTRAN.

Iostream library: Describes the components needed for C++ input/output operations.

C library: Included in C++ primarily by reference. Some modifications were needed to ensure that the functions provided follow the rules required by C++ for static type safety. The description of the C library is sprinkled throughout the C++ Standard library sections; each C library component is associated with the C++ library components that provide related functionality.

The components offered by the C++ Standard library are extensive. With this variety, applications in a wide variety of domains can be written to be portable to all implementations supporting the C++ Standard library.

Schedule for the C++ Standard

When are we going to have a C++ standard? Since September 1994, the document that will eventually become the C++ standard has been passing through various phases required by ANSI and ISO before the document can be called a standard. The technical work of the committee should be completed by the end of 1995, and the C++ Standard published by the end of 1996.

In September 1994, the document was sent to ISO for the Committee Draft (CD) registration ballot. ISO distributed the document to the participating member countries, and the C++ technical experts of each country evaluated the document based on the following two criteria:

- Does the document describe a complete set of language features?
- Does the document describe sufficient support for class libraries?

The ballot was successful, meaning that the language and library features were frozen and the C++ Standard committee could no longer accept any new language extensions or class libraries.

In March 1995, the document (revised according to comments received during the CD registration ballot) was sent to ISO for the publication of the Committee Draft. At this point, the member countries sent out the document for public reviews. ANSI, the U.S. representative to ISO, began its public review period on May 26, 1995. The goal of this review is to decide whether the description of the language and library features is clear and unambiguous and whether the document can become an international standard. The ANSI public review period is scheduled to end on July 26, 1995. ISO must receive the position of each member country by the middle of August 1995, and then determine whether the document is technically good enough and clear enough to become an international standard.

After that happens, the committee must review the document and clarify some sections according to the public comments received. Four to eight months are scheduled for this work.

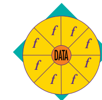
The committee should be ready to send the document to ISO for the last phase of revisions at the beginning of 1996. During this review, ISO will verify the format of the document to make sure it meets all the ISO requirements for an international standard document. The member countries will be asked to review the document one last time—for typographical errors, font errors, and so forth. The results of this last review will be known by the end of 1996. If the ballot is successful, the international standard for the C++ programming language will then be published.

Acknowledgment

The information provided in this article has appeared in other forms in the "Standard C++" columns I write for the *C++ Report* magazine published by SIGS Publications. Stan Lippman, the editor, has kindly agreed to let me reuse the material for this article.

References

1. Ellis, M. A. and Stroustrup, B. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.
2. International Standard for Information Systems. *Programming Language C*. ISO/IEC 9899:1990.



Josee Lajoie, IBM Canada Ltd., 1150 Eglinton Ave East, North York, Ontario, Canada, M3C 1H7. Internet: josee@vnet.ibm.com. Ms. Lajoie is a staff development analyst in the VisualAge C++ Compiler group. She is vice-chair of the ANSI/ISO C++ Sandard committee and the chair of the Core Language Working group for the committee. She writes the Standard C++ columns for the C++ Report magazine. Ms. Lajoie received a BEng in Electrical Engineering from L'Ecole Polytechnique of the University of Montreal.