C/C++ Users Journal™
Advanced Solutions for C/C++ Programmers

What's New in Standard C++

by Chuck Allison

Standard C++ is finally real, after nine years in the making. Chuck supplies a quick guided tour of the end result.

It's official! On July 20, 1998, all 20 ISO national bodies involved in the work of standardizing the C++ programming language approved the document the committee had created and maintained since 1990. This may not seem like a big deal to the casual observer, but I shudder to contemplate the cost of producing that dense, almost 800-page document.

When I joined the committee in March 1991, one year after the technical work had begun, the goal was to hurry up and ''get this thing done,'' which at the time meant, ''add templates and exceptions to the language, tighten up IOStreams, and maybe add a string class.'' Were we naïve or what? There was such demand for more robust support of object-oriented programming, containers, and programming in the large that by 1994 we had also added run-time type identification, the Standard Template Library, and namespaces, some of which represented more invention than the standardization of existing practice. The committee needed the next three years just to work the bugs out. The result is an incredibly powerful and elegant, if not sometimes overwhelming, programming language.

In this article I summarize what has changed since AT&T's C++ V2.0, which is what most of us were using before the standards committee went to work. To keep things orderly, I follow (most of) a traditional outline:

• Something Old — deprecated features and what has changed in existing features

• Something New — features that have been added to the language

without much precedent from other languages, such as Run-time Type Identification (RTTI), covariant return types, and new-style casts

• Something Borrowed — includes type **bool**, templates, exceptions, and namespaces, since they were patterned after features found in other languages.

In a future article I'll discuss some of the newer features of the Standard C++ library [something blue? — pjp].

Keep in mind that compilers still do not support all language features (they're only implementers, after all :-). For this article, I'm using version 2.39 of the Edison Design Group (EDG) demo compiler for Windows NT with the Dinkum C++ Library (V2.2), which I believe is the most up-to-date combination available. (Only one of the following examples, template-template arguments, fails to compile. Your mileage may vary).

Something Old

If you've been using C++ for a while, you need to be aware of changes to certain language features. Theoretically, you may need to recompile some of your code, but most compilers will likely provide a legacy mode to compile existing code. Notable changes follow.

Banning implicit **int** *in declarations*

If you've ever let your eyes pore over vintage C code, you've probably seen a large number of holes where the **int** keyword should have appeared. That's because C infers **int** as a type name in many instances (as a favor to lazy Unix hackers, I suppose :-). The following declarations are now invalid in C++ (and will be in C9X also):

```
const n = 7;
void f(const n) {.....}
g() {.....}
```

If you want a program that works with a conforming compiler, you need to explicitly use the **int** keyword in each of these contexts, as in:

```
const int n = 7;
void f(const int n) {.....}
int g() {.....}
```

A noticeable consequence of this change is that you must also explicitly declare **main** as a function returning **int**, in one of the following ways:

```
int main() {.....}
int main(int argc, char* argv[])
{.....}
```

You don't have to explicitly return a value from **main**, however. If you

don’t, it’s the same as if you had returned a zero.

Default behavior of **operator new**

Instead of returning a null pointer, **operator new** now throws an exception when a dynamic memory allocation request fails, as the following snippet illustrates:

```
#include <new>

int main()
{
    int* p;
    try {
        p = new int[1000000L];
    }
    catch (std::bad_alloc&) {
        // memory failure
    }
}
```

The **bad_alloc** exception class is defined in the header **<new>** as a member of namespace **std**, which contains most of the declarations in the Standard C++ library. (More about namespaces later.)

There is a special version of **operator new** for those times when you want to revert to the traditional null-return behavior, which goes like this:

```
#include <new>

int main()
{
    int* p =
        new (std::nothrow) int[1000000L];
    if (!p)
        ..... // failed
}
```

Scope of **for**-*init declarations*

Most C++ programmers are probably accustomed to declaring loop index variables in the init-part of a **for** loop, such as:

```
for (int i = 0; i < n; ++i)
{.....}
int j = i;   // error!
```

What is less well known, most likely because compilers haven’t supported it, is that the scope of the variable **i** above is the body of the loop only.

Static const initializers

This change to the language won’t require re-compilation, but offers a valuable convenience. You can now initialize static const data members within the class definition itself. Not only is this a boon to readability, but

it lets you use such variables in subsequent member declarations, like array dimensions, replacing the ubiquitous enum hack:

```
class Foo
{
    static const int MAXOBJS = 100;
    static const int numObjs = 0;
    static Foo objects[MAXOBJS];
};
const int Foo::numObjs;
const int Foo::MAXOBJS;
Foo Foo::objects[MAXOBJS];
```

As this code illustrates, you still have to define the space outside the class definition, and you mustn't repeat the initialization there.

Deprecated Features

A deprecated or obsolescent language feature is one that may disappear due to future standards work. What this really means is that the feature is either no longer needed, or is undesirable for one reason or another. The 1989 standard for C, for example, marked old-style function definitions (i.e., without prototyping) as obsolescent. I wouldn't be surprised if the C9X committee votes to drop them from the language this time. The newly approved C++ Standard has made the following language features obsolescent:

Invoking Standard C headers with a .h suffix

We traditionally think of headers as files, but a compiler is free to make a header's declarations available in any manner it chooses. To encourage this point of view, the C++ standards committee voted rather early to drop the .h suffix for C++ headers. This means that you should write:

```
#include <iostream>
```

instead of:

```
#include <iostream.h>
```

although most compilers will allow both. When namespaces were added to the language, the committee decided to wrap most C++ and all Standard C library declarations in the namespace **std**, and to rename the C headers by prepending a ''c'' and dropping the . h suffix. This means that the preferred method of getting at C library features is the same for using C++ library elements. For example:

```
#include <cstdio>
int main()
{
    std::printf("hello, world\n");
}
```

Thinking that this might be too much of a culture shock, the committee

decided to deprecate instead of disallow altogether the traditional .h header. For now, if you say:

```
#include <stdio.h>
```

it's as if you had written:

```
#include <cstdio>
```

followed by a using declaration for each identifier defined in the header (so you don't have to use the **std::** prefix).

Old-style casts

Old C-style casts are dangerous and ugly, but sometimes a programmer's got to do what a programmer's got to do. Unfortunately, programmers occasionally do the wrong thing. The new-style C++ casts are superior to C-style casts because they:

• explicitly advertise the type of cast being performed

• disallow any type of conversion other than the one requested, and

• stand out in code inspections because of their noticeable syntax

For example, the expression:

```
p = reinterpret_cast<char*> (0x00f0c10a)
```

is much more likely to draw a reader's attention than:

```
p = (char*)(0x00f0c10a)
```

Furthermore, using **static_cast** here instead of **reinterpret_cast** would fail, since the former converts only between related types. For now, old-style casts are only deprecated, since otherwise too much existing code would break, but watch out! Five years from now they're probably going away.

Static declarations at file scope

The old-fashioned way of making an identifier private to its translation unit is to declare it static at file scope:

```
static void f() {.....}
```

The modern way to organize identifiers is inside namespaces. Anything you declare at file scope is part of the ‘‘global namespace,’’ which is visible across compilation units. If you want to hide identifiers declared outside of any block from other compilation units, you should place them in the *unnamed namespace*, as follows:

```
namespace
{       // has no name
    void f() {.....}
};
```

Each unnamed namespace in a compilation unit behaves as if it has a name unique to that compilation unit, so there is no possibility that you can access its identifiers from any other file. Since there is no name for this namespace, there is no way to define a function outside of the namespace body, as you typically do with the scope resolution operator for member functions and functions in named namespaces.

Access declarations

When using less than public inheritance, the access of inherited members decays to the level of the inheritance used, if applicable. For example, in the following code excerpt, without the access declaration **B::f** in the public section of **D**, **f** would not be accessible to clients of **D** objects.

```
class B
{
public:
    void f();
};

// private inheritance makes f
// private in D, hence inaccessible
// to D's clients
class D : private B
{
public:
    B::f; // old-style access
          // declaration
};
```

Old-style access declarations are now deprecated in favor of using declarations, which were introduced with namespaces. In the case of class **D** above, you should instead write:

```
class D : private B
{
public:
    using B::f;
};
```

<strstream.h> classes

The classes in <strstream.h> are C++ equivalents of the functionality provided by the Standard C library functions **sscanf** and **sprintf**, which support in-core formatting of C-style strings. In old C++, you would do something like the following to build a null-terminated string via output operations:

```
#include <iostream.h>
#include <strstream.h>
```

```
main()
{
    ostrstream os;
    os << "A number: " << 7 << ends;
    char* s = os.str();
    cout << s << endl;
    os.rdbuf()->freeze(0);
}
```

The resulting output is:

```
A number: 7
```

The **str** member function yields a **char*** that points to the dynamically constructed array of characters. You can insert the **ends** manipulator to get a terminating null. Since it uses the heap to build the string, the object **os** leaves you responsible for deleting the memory when you're through, unless you give ownership back to **os** via the **strstreambuf::freeze** member function. There is also an **istrstream** class for doing stream input on a **char** array.

Standard C++ has deprecated the use of these classes, and, like the rest of the library, they now reside in the **std** namespace in the appropriately named header <strstream>. If you want to do things the Right Way, however, you will now use the **ostringstream** and **istringstream** classes defined in <sstream>, which work on instances of **std::string**, instead of arrays of characters, as the following example illustrates:

```
#include <iostream>
#include <sstream>

main()
{
    std::ostringstream os;
    // no ends
    os << "A number: " << 7;
    std::string s = os.str();
    std::cout << s << std::endl;
    // no freeze(0)
}
```

You don't need the **ends** manipulator because C++ strings don't need terminating nulls. Since the **str** function gives you a pointer to a copy of the string, **ostringstream** maintains ownership of any heap memory it uses, rendering a ''freeze'' unnecessary.

Something New

The following features have been manufactured ''out of whole cloth'' during the last five years.

Explicit constructors

Single-argument constructors routinely provide implicit conversions from one type to another. For example, mathematical classes such as **Complex**

typically have a constructor with the signature **Complex(double)**, as well as global functions for arithmetic operations, such as **Complex operator+(const Complex&, const Complex&)**. With these definitions in place, you can write mathematical expressions like you would by hand, such as:

```
Complex c1(1, 2);
Complex c2 = c1 + 1;  // c1 + (1, 0)
```

The compiler implicitly converts the literal **1** to **Complex(1.0)** to accommodate **operator+(const Complex&, const Complex&)**. To give the programmer control over when implicit conversions apply, Standard C++ provides the **explicit** keyword. An explicit constructor is never used for implicit conversions. If you define the single-argument constructor above like this:

```
explicit Complex(double);
```

then the addition operation in the initialization of **c2** above fails. And in case you're wondering, no, you can't use **explicit** with conversion operators, although it would make sense to do so.

Mutable data members

It is widely accepted that the constness of member functions should always be determined from a client perspective, but sometimes you want to update some hidden data members the user knows nothing about during an apparently const operation. Traditionally, you would ‘‘cast away’’ the constness of the **this** pointer to gain access to those data members. For example, the lookup method in the **List** class below uses the **const_cast** operator to gain access to the cache:

```
class List
{
    void* cache;
public:
    bool lookup(void* p) const
    {
        // Do lookup, then cache
        // pointer for next time
        const_cast<List*>(this)-> cache = .....;
    }
};
```

Although you may have other reasons for casting away const, this particular situation occurs so frequently that C++ now allows you to declare data members **mutable**, so you can modify them directly in a const member function:

```
class List
{
    mutable void* cache;
public:
    bool lookup(void* p) const
    {
```

```
        // Do lookup, then cache
        // pointer for next time
        cache = .....;
    }
};
```

Covariant returns

A long-standing C++ rule requires that a member function that overrides a virtual function must have not only the same signature but also the same return value as the base class member function. In the following code, for example, **A::f** and **B::f** both return a pointer to an object of class **X**:

```
class X {};

class A
{
public:
    virtual X* f() {return new X;}
};

class B : public A
{
public:
    virtual X* f() {return new X;}
};
```

In real-world object models, however, it is quite common for **B::f** to want to return a pointer to an object derived from X. Standard C++ allows such covariant returns, so you can modify the code as follows:

```
class Y : public X {};

class B : public A
{
public:
    virtual Y* f() {return new Y;}
};
```

New-style casts

As I indicated earlier, C-style casts have long been a source of controversy as well as bugs. Since they represent a work-around to normal language behavior, they should be used sparingly, and should be easy to spot in code. To this end, the committee invented new-style casts, which come in four flavors: **static_cast**, **dynamic_cast**, **reinterpret_cast**, and **const_cast**.

static_cast is for converting between related types, such as numeric types, as in:

```
double x;
// truncate x
int i = static_cast<int>(x);
```

You can also use **static_cast** to downcast from a pointer to base to a pointer to a derived object:

```
X* px = new Y; // Y derives from X,
               // as in 3 above
Y* py = static_cast<Y*>(px);
```

This particular use of **static_cast** is safe only when you know at compile time that **px** actually points to a **Y** object. If you don't, you can use **dynamic_cast** to find out for sure:

```
Y* py = dynamic_cast<Y*>(px);
if (py)
    .....  // actually points to a Y
else
    .....  // doesn't point to a Y
```

dynamic_cast works only for polymorphic types (classes that have a virtual function) and built-in types. You can also use **dynamic_cast** for casting reference types, in which case the cast throws a **bad_cast** exception if the referent isn't what you expected. You'll find the definition for **bad_cast** in the header <typeinfo>. I showed examples of the other two casts earlier in this article.

Run-time type identification (RTTI)

dynamic_cast is part of RTTI, the C++ mechanism that lets you query the dynamic type of an object through a pointer or reference. There is also a **typeid** operator that yields an object of class **typeinfo**, which has limited information about an object's dynamic type. The following example shows how to get the name of an object's dynamic type:

```
#include <typeinfo>

main()
{
  D d;
  B* bp = &d;

  cout << typeid(d).name() << endl;
  cout << typeid(bp).name() << endl;
  cout << typeid(*bp).name() << endl;
  cout << (typeid(d) == typeid(*bp))
       << endl;
}
```

This example should yield output something like:

```
D
B *
D
1
```

Needless to say, you shouldn't use RTTI very often in typical applications. Unless you're writing a utility that needs to specifically query an object's dynamic type, such as a debugger, polymorphism via virtual functions should meet your dynamic binding needs.

Something Borrowed

It may seem strange to label the major new features of Standard C++ ‘‘borrowed,’’ but one must give credit where it is due. The following table lists these features and the languages that inspired them.

```
Feature     Borrowed From

type bool   Pascal and many others
namespaces  Lisp/CLOS, Modula 2, Ada
exceptions  Lisp, ML
templates   Ada, Clu
```

A **bool** variable holds the result of a Boolean expression. C++ also provides the Boolean literals **true** and **false**, should you need them. When used in an integer expression, **true** becomes 1 and **false** becomes zero, so you can use them as array indices if you want. There is even an output stream flag, **boolalpha**, that causes a stream such as **cout** to print the string literals **"true"** and **"false"** instead of 1 and 0 for corresponding Boolean values. The following example illustrates all these features:

```
#include <iostream>
#include <string>
using namespace std;

// test an int’s parity
bool odd(int n)
{
    return n%2 == 1;
}

string parity[] = {"even", "odd"};

int main()
{
    int n = 7;
    cout.setf(ios::boolalpha);
    cout << "n odd? " << odd(n)
         << endl;
    cout << "parity of n-1 == "
         << parity[odd(n-1)] << endl;
}
```

This produces the output:

n odd? true parity of n-1 == even Namespaces

C++ lets you partition your declarations into namespaces — to package them together, so to speak. It’s essentially the C++ equivalent of modules in Modula-2, or packages in Ada, Common Lisp, and Java. The main reason for namespaces is to manage ‘‘programming in the large’’ by minimizing name conflicts that might occur when using multiple libraries in a program. Library vendors have traditionally invented strange looking names for global identifiers to uniquely identify them. Often this has involved some prefix related to the name of the company or particular library product. Most of Rogue Wave’s library functions, for example,

begin with the characters **RW**. With namespaces, you don't worry about name conflicts (other than the names of namespaces themselves) because declarations remain hidden within a namespace until you ask for them. To define a namespace, you use the **namespace** keyword, like this:

```
namespace MyNamespace
{   // namespace begins here
    void f();
    // more declarations .....
};      // namespace ends here
```

The full name of the function **f** above is **MyNamespace::f**, similar to a class member. If you refer to **f** with its fully qualified name, therefore, there is no chance for a conflict with any other function named **f**. If **MyNamespace::f** is the only function named **f** that you will be using in a given scope, then you can say so with a *using declaration*, and then just use **f** unadorned thereafter to save keystrokes, as in:

```
using MyNamespace::f;
f();  // calls MyNamespace::f
```

If you're confident that none of the names in **MyNamespace** conflict with other names in your program, you can import all of the names with a single *using directive*:

```
using namespace MyNamespace;
```

A using directive in essence ‘‘unlocks’’ an entire namespace, so that all of its names are considered when the compiler seeks to match a use of an identifier with its original declaration. You encounter a problem only when you use a name that is declared in more than one place — its mere presence in multiple namespaces is not a liability, as is the case with traditional link libraries. It is generally bad practice, however, to put using directives in a header file, since any translation unit that includes a directive will unlock the associated namespace, thus defeating the purpose of having namespaces. The same logic applies to individual using declarations, as well (only on a smaller scale, since you're exposing identifiers only one by one). The rule of thumb, therefore, is to use only fully qualified references to namespace members in your own header files.

Declarations for the same namespace can occur in different header files. The complete namespace is the union of the declarations included in a translation unit. This is how the standard namespace **std** is defined. Each standard header wraps its declarations in the definition:

```
namespace std {.....};
```

but there is only one namespace **std**.

If you don't like typing out long namespace names, you can define an alias, like this:

```
namespace my = MyNamespace;
```

You can thereafter refer to namespace members with the alias, e.g., **my::f()**. This becomes a cheap versioning tool as well, since you can precede code that assumes a particular namespace name (like **my** above) with an alias declaration that you can change at will.

Exceptions

C++ exceptions support runtime error handling in a robust manner. Exceptions behave somewhat like the **setjmp/longjmp** mechanism of Standard C, only they are much safer and more flexible. To throw an exception, you use the **throw** keyword:

```
if (<something bad happens>)
    throw MyException("hit the fan");
```

To handle this exception, there must be a try block with an associated catch clause that can catch a **MyException** object, somewhere back up the thread of execution. For example:

```
try
{
    f();
}
catch (MyException& x)
{
    cout << x.what() << endl;
}
```

The Standard C++ library defines several exception classes for the objects it throws. Much has been written on exceptions, so to save space here I'll just refer you to my two-article series, ''Error Handling in C++,'' *CUJ*, November-December 1997, or to chapter 13 of my book, *C & C++ Code Capsules* (Prentice-Hall, 1998), which derives from those articles.

Templates

The crowning piece of work of the C++ standards committee is most assuredly the template mechanism. What started out as merely a formalism for generating type-parameterized code has become a programming paradigm in its own right. As proof, just consider the power, flexibility, and popularity of the Standard Template Library. Mike Vilot, past chair of the committee's library group, once called STL ''templates on steroids.'' STL has generated a subculture of its own, and has even spilled over into Java as JGL (the ''Java Generic Library''), which was developed by ObjectSpace, one of the first vendors of STL.

Templates come in two varieties: class and function templates. Both occur throughout the Standard C++ library. In fact, the Standard C++ library is almost 100% templates. For example, the **string** class is really an instantiation of the **basic_string** template for **char** string elements. **cout** is an object of type **ostream**, which in turn is an instantiation of the

template class **basic_ostream**, for **char** stream elements. How about **complex**? Yep, it's a template, with specializations for **float**, **double**, and **long double** right out of the box. The only notable library components I can think of off the top of my head that aren't templates are the basic library support functions, such as **set_terminate** and the various overloads of **operator new** and **operator delete**.

Class templates seem to have evolved from the need to build code once for a container and then adapt it for use on different types of contained objects. Imagine a container of integers, such as the following (substitute **stack** or **set** or some other favorite container in place of **Container**):

```
class Container
{
    int *data; // uses an array
    int n;     // #elements
public:
    Container();
    void insert(int);
    void remove(int);
    bool contains(int) const;
    int* first();
    int* next();
};
```

The logic of the member functions is the same whether a **Container** holds **int** elements or **float**, or objects of any user-defined type. To capture that fact, templates let us make the type of the contained object a parameter:

```
template<class T>
class Container
{
    T *data;   // use an array
    int n;     // #elements
public:
    Container();
    void insert(const T&);
    void remove(const T&);
    bool contains(const T&) const;
    T* first();
    T* next();
};
```

When you need a **Container** of integers, you specialize the template by specifying the actual type for its parameter, as follows:

```
Container<int> c;
```

whereupon the compiler generates the appropriate code from the template.

Function templates behave differently, in that the compiler deduces the type of the template parameter(s) from the arguments to the function call. For example, given the template:

```
template<class T>
```

```
void swap(T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

the call **swap(i, j)**, where **i** and **j** both have type **int**, will cause code for the **int** version of **swap** to be generated automatically at compile time. Compilers have supported basic class and function templates for some time now.

That's the easy explanation of templates. In the rest of this section I summarize features that have evolved over the years that make templates the sophisticated tool for generic programming that they have become.

Non-type parameters

Template parameters don't have to be types. They can also be compile-time constants, such as an integer for array limits. For example, if you know an upper bound for the number of elements a container in your application will hold, then you can store the elements more efficiently inside the container object, as follows:

```
template<class T, size_t N>
class Container
{
    T data[N]; // the elements
    int n;
public:
    Container();
    void insert(const T&);
    void remove(const T&);
    bool contains(const T&) const;
    T* first();
    T* next();
};
```

The declaration:

```
Container<int, 100> c;
```

defines the data array for **c** to occupy 100 words within the object.

Default arguments

You can provide default values for both type and non-type template parameters. For example, if you change the template parameter specification for **Container** above to:

```
template<class T, size_t N = 100>
class Container {.....}
```

then the declaration:

```
Container<int> c;
```

fixes the capacity of **c** at 100, as in the previous example. For a type parameter example, consider the **set** container type in the Standard C++ library. The declaration:

```
set<int> s1;
```

causes s1 to store its elements in ascending numerical order because the **set** template is defined as:

```
template<class T, class Compare = less<T>, .....>
class set {.....};
```

The construction of a **set** object involves the creation of a function object of type **Compare**, which is of type **less<T>** by default. The container uses this function object to compare elements by pairs to maintain proper ordering for the set. To store elements in descending order, you can provide a different function object class, like this:

```
set< int, greater<int> > s2;
```

The specification of **Compare** shows that you can use a template parameter (**T** in this case) to define subsequent parameters in the same template definition (**less<T>**).

Member templates

You can define a template as a class member function, for example

```
class A
{
    template<class T> void f(const T& t);
    // .....
};
```

This is especially useful for implicit conversions between related template classes. As I mentioned earlier, the Standard C++ library defines a **complex** template class, with three specializations for the types **float**, **double**, and **long double**. To allow conversions between these various precisions, the following member template is defined:

```
template<class T>
class complex
{
public:
    template<class X> complex(const complex<X>&);
    // .....
};
```

To define the function body outside the class definition, you have to mention both template parameters, as follows:

```
// Definition:
template<class T1>
    template<class T2>
```

```
    complex<T1>::complex(const complex<T2>& c)
    {.....};
```

The member constructor template above is used in the construction of **c2** below:

```
// Usage:
complex<float> c1;
complex<double> c2(c1); // uses template ctor
```

Explicit instantiation

A C++ compiler decides which template code to generate by the objects you declare and the functions you call. When calling a function template, however, the arguments must match the template definition exactly. To illustrate, suppose you write a template to find the maximum of two integers:

```
template<class T>
T max(const T& t1, const T& t2)
{
    return t1 > t2 ? t1 : t2;
}
```

In the following function call the template will not apply:

```
int i = 1;
double x = 2.0;
cout << max(i,x); // error
```

The reason this fails is because the template expects the arguments to be the same type. You can instruct the compiler to generate the **double** version ahead of time, however, in which case the standard conversion from **int** to **double** can be used to favor this keyword. To do this, just declare a version of the function with the **template** keyword, as in:

```
template double max(double, double);
```

This explicit instantiation brings the **double** version of **max** into existence as if you had coded it yourself. You can explicitly instantiate template classes as well.

Explicit specification of template functions

It is possible to define a function template that does not necessarily use all of its template parameters in its arguments. The following example, while not terribly useful, illustrates the point:

```
template<class T>
T* build()
{
    return new T;
}
. ....
int* ip = build<int>();
```

Since there is no way to deduce the type of **ip** from the call to **build**, you need a way to specify it explicitly. A more useful example, which almost made its way into the Standard C++ library but didn't for lack of time, is **implicit_cast**, a function template that looks like a new-style cast:

```
template<class To, class From>
To implicit_cast(const From& f)
{
    return f;
}
. ....
int i = 1;
double x = implicit_cast<double>(i);    // int deduced
char* p = implicit_cast<char*>(i);      // conversion error
```

This pseudo-operator allows you to force an implicit conversion. I know that sounds strange, but **implicit_cast** succeeds only if an implicit conversion naturally exists between the two types involved, which sometimes is just what you want. The only way to introduce the target type is with the explicit specification syntax, which is by coincidence the same as the new-style cast syntax. (New-style casts are bona fide operators, though, and not templates).

Explicit specialization

You may sometimes want to override the code-generation mechanism for special cases. Consider a function much like **strcmp** that compares objects:

```
// A Comparison function template
template<class T>
int comp(const T& t1, const T& t2)
{
    return (t1 < t2) ? -1 : (t1 == t2) ? 0 : 1;
}
```

This works fine for value-oriented types that support **operator<** and **operator==**, but certainly not for pointer types like **char***. You can specialize this template for **const char*** to compare null-terminated strings instead of the pointers themselves, as follows:

```
// A char* specialization:
template<>
int comp<const char*>(const char*& t1, const char*& t2)
{
    return strcmp(t1,t2);
}
```

The **template<>** prefix signifies that the code that follows is a full specialization of a previously defined template. Since in this case the compiler could deduce that **T** is **const char***, you could omit the **<const char*>** specification after the function name. For that matter, in this example you could ignore template syntax altogether and just define a function named **comp** that takes **const char*** arguments, since the compiler considers ordinary functions before templates to match a

function call. But it is a good idea to do it like I did here, so the reader knows that it overrides a template. And of course you must use the template syntax when the function's return type is a template argument, or if you are specializing a class template.

Partial specialization

For class templates with multiple arguments, you can choose to specialize only some of those arguments. You can easily spot such partial specializations by the occurrence of type specifications both in the template prefix and immediately after the class name. To illustrate, consider the following primary template with two type parameters:

```
template<class T, class U>
class A
{
public:
    A() {cout << "primary template\n";}
};
```

The following partial specialization overrides the primary template when **T** is a pointer type.

```
template<class T, class U>
class A<T*, U>
{
public:
    A() {cout << "<T*, U> specialization\n";}
};
```

The next one applies when the types of the template arguments are the same.

```
template<class T>
class A<T, T>
{
public:
    A() {cout << "<T, T> specialization\n";}
};
```

The following specializes the case where **T** is **int**.

```
template<class U>
class A<int, T>
{
public:
    A() {cout << "<int, U> specialization\n";}
};
```

The following test program verifies the above statements:

```
int main()
{
    A<char, int> a1;
    A<char*, int> a2;
    A<float, float> a3;
    A<int, float> a4;
```

```
}
```

It should produce the output:

```
primary template
<T*, U> specialization
<T, T> specialization
<int, U> specialization
```

Partial specialization comes in handy when you want to avoid the code bloat that sometimes results with templates. As you know, the compiler generates separate code for each type you specialize a class template for. For example, the declarations **Container<int>**, **Container<float>**, and **Container<Foo>**, generate three distinct class definitions. Using partial specialization you can arrange for all pointer types to share a common implementation. First, fully specialize on **void***:

```
// The Primary Template:
template<class T>
class Container {.....};

// Full Specialization:
template<>
class Container<void*> {.....};
```

Then, partially specialize on **T***, as follows:

```
// Partial Specialization:
template<class T>
class Container<T*> : private Container<void*>
{.....};
```

The **T*** specialization uses the **void*** implementation, casting **T* to and from void*** as needed. Whenever you specialize **Container** for a pointer type other than **void***, the **T*** specialization is used, so you don't have a different instantiation for each pointer type.

Template-template arguments

No, I'm not stuttering :-). You can actually use another template as a template argument. The following should suffice as an illustration:

```
template<class T, template<class U> class Container>
class SomethingBig
{
    Container<T> c;
    .....
};
```

In addition to whatever else it may do, **SomethingBig** allows you to arbitrarily specify a container type with its associated contained type, and it builds the container in its implementation. You actually pass the container template as an argument, like this:

```
SomethingBig<int, vector> sb_vec_int
SomethingBig<Foo, map> sb_map_Foo;
```

Since the formal parameter **U** is not actually used, you can omit it, just like you can omit the names of formal parameters in C++ function definitions:

```
template<class T, template<class> class Container>
class SomethingBig {.....};
```

typename

There are some obscure instances in template definitions that can really confuse a compiler (and you thought you were confused! :-). For example, the expression **T::U** inside a template definition with a type parameter named **T** could represent either a typedef member or a static data member of type **T**, and compilers aren't sufficiently clairvoyant to distinguish between the two. The **typename** keyword cues the compiler that the following token is a type, as the following definition of **Baz** illustrates:

```
template<class T>
class Baz {
    typename T::U x;
};
```

You can also use **typename** in place of the **class** keyword in specification of template parameters:

```
template<typename T>
    .....
```

Conclusion

As you can see, C++ has evolved significantly since its early days at AT&T Bell Labs. Some things have changed in the name of type safety, and other features have been added to support more sophisticated programming techniques. A lot has been said about the language over the years. One particular programming language has been marketed in large part as a supposedly simpler alternative to C++.

Is C++ complex? When taken as a whole, perhaps. Is it worth the effort to master? I think so, although you can use it effectively without ‘‘mastering’’ it. In any case, perhaps the most important thing that could be said at this point is that C++ is stable. It is widely used, and now it has an ISO/ANSI standard.

The future looks bright for C++.

About the Author

Chuck Allison is Consulting Editor and a former columnist with *CUJ*. He is the owner of Fresh Sources, a company specializing in object-oriented software development, training, and mentoring. He has been a

contributing member of J16, the C++ Standards Committee, since 1991, and is the author of *C and C++ Code Capsules: A Guide for Practitioners*, Prentice-Hall, 1998. You can email Chuck at cda@freshsouces.com.

Home | **Top** | **Search**