C++ Tutorials

C++ Language and Library

- Namespaces
- New Fundamental Type - bool
- Stream I/O
- Virtual Functions
- Templates
- Use of Static
- Mutable
- Explicit
- Standard Template Library
- Exception Handling
- Placement New/Delete
- Pointers to Members and Functions
- Type Identification
- Dynamic Casts
- Explicit Template Argument Specification
- Using Standard Libraries
- Operators new[] and delete[]
- C++ Character Sets
- Allocators

C++ Namespaces

INTRODUCTION TO NAMESPACES - PART 1

Namespaces are a relatively new C++ feature just now starting to appear in C++ compilers. We will be describing some aspects of namespaces in subsequent newsletters.
What problem do namespaces solve? Well, suppose that you buy two different general-purpose class libraries from two different vendors, and each library has some features that you'd like to use. You include the headers for each class library:

```
#include "vendor1.h"
#include "vendor2.h"
```

and then it turns out that the headers have this in them:

```
// vendor1.h

... various stuff ...

class String {
    ...
};

// vendor2.h

... various stuff ...

class String {
    ...
};
```

This usage will trigger a compiler error, because class String is defined twice. In other words, each vendor has included a String class in the class library, leading to a compile-time clash. Even if you could somehow get around this compile-time problem, there is the further problem of link-time clashes, where two libraries contain some identically-named symbols.
The namespace feature gets around this difficulty by means of separate named namespaces:

```
// vendor1.h

... various stuff ...

namespace Vendor1 {
    class String {
        ...
    };
}
```

```
// vendor2.h

... various stuff ...

namespace Vendor2 {
    class String {
        ...
    };
}
```

There are no longer two classes named String, but instead there are now classes named Vendor1::String and Vendor2::String. In future discussions we will see how namespaces can be used in applications.

INTRODUCTION TO NAMESPACES - PART 2

In the last issue we discussed one of the problems solved by namespaces, that of conflicting class names. You might be using class libraries from two different vendors, and they both have a String class in them. Using namespaces can help to solve this problem:

```
namespace Vendor1 {
    class String { ... };
}

namespace Vendor2 {
    class String { ... };
}
```

How would you actually use the String classes in these namespaces? There are a couple of common ways of doing so. The first is simply to qualify the class name with the namespace name:

```
Vendor1::String s1, s2, s3;
```

This usage declares three strings, each of type Vendor1::String.
Another approach is to use a using directive:

```
using namespace Vendor1;
```

Such a directive specifies that the names in the namespace can be used in the scope where the using directive occurs. So, for example, one could say:

```
using namespace Vendor1;

String s1, s2, s3;
```

and pick up the String class found in the Vendor1 namespace.
What happens if you say:

```
using namespace Vendor1;
using namespace Vendor2;
```

and both namespaces contain a String class? Will there be a conflict? The answer is "no", unless you try to actually use String. The using directive doesn't add any names to the scope, but simply makes them available. So usage like:

```
String s1;
```

would give an error, while:

```
Vendor1::String s1;
```

```
    Vendor2::String s2;
```
would still work.
You might have noticed that namespaces have some similarities of notation with nested classes. But namespaces represent a more general way of grouping types and functions. For example, if you have:
```
    class A {
        void f1();
        void f2();
    };
```
then f1() and f2() are member functions of class A, and they operate on objects of class A (via the "this" pointer). In contrast, saying:
```
    namespace A {
        void f1();
        void f2();
    }
```
is a way of grouping functions f1() and f2(), but no objects or class types are involved.

INTRODUCTION TO NAMESPACES - PART 3

In previous issues we talked about how C++ namespaces can be used to group names together. For example:
```
    namespace
A {
        void f1();
        void f2();
    }

    namespace B {
        void f1();
        void f2();
    }
```
The members of the namespace can be accessed by using qualified names, for example:
```
    void g() {A::f1();}
```
or by saying:
```
    using namespace A;
    void g() {f1();}
```
Another interesting aspect of namespaces is that of the unnamed namespace:
```
    namespace {
        void f1();
        int x;
    }
```
This is equivalent to:
```
    namespace unique_generated_name {
        void f1();
        int x;
    }
    using namespace unique_generated_name;
```

All unnamed namespaces in a single scope share the same unique name. All global unnamed namespaces in a translation unit are part of the same namespace and are different from similar unnamed namespaces in other translation units. So, for example:

```
namespace {
    int x1;
    namespace {
        int y1;
    }
}

namespace {
    int x2;
    namespace {
        int y2;
    }
}
```

x1 and x2 are in the same namespace, as are y1 and y2.
Why is this feature useful? It provides an alternative to the keyword "static" for controlling global visibility. "static" has several meanings in C and C++ and can be confusing. If we have:

```
static int x;
static void f() {}
```

we can replace these lines with:

```
namespace {
    int x;
    void f() {}
}
```

INTRODUCTION TO C++ NAMESPACES - PART 4

Previously we talked about how namespaces can be used to group names into logical divisions:

```
namespace Vendor1 {
    class String {
        ...
    };
    int x;
}

namespace Vendor2 {
    class String {
        ...
    };
    int x;
}
```

and how those names can be accessed via qualification:

```
Vendor2::String s;
```

or a using directive:

```
using namespace Vendor2;

String s;
```

Another way of accessing names is to employ a using declaration:

```
using Vendor2::String;

String s;
```

This can be a little confusing. A using directive:

```
using namespace X;
```

says that all the names in namespace X are available for use, but none of them are actually declared or introduced. A using declaration, on the other hand, actually introduces a name into the current scope. So saying:

```
using namespace Vendor2;
```

makes String and x available for use, but doesn't declare them. Saying:

```
using Vendor2::String;
```

actually introduces Vendor2::String into the current scope as a declaration. Saying:

```
using Vendor1::String;
using Vendor2::String;
```

will trigger a "duplicate declaration" compiler error.
There are several other aspects of using declarations that are worth learning about; these can be found in a good C++ reference book.

New Fundamanental Type - bool

A new fundamental (builtin) type has recently been added to C++. It is a type for representing Boolean values and uses the keyword "bool". For example, you could say:

```
bool b;

b = true;
if (b)
    ...
```

A bool value is either true or false. A bool value can be converted to an integer:

```
bool b;
int i;

b = false;
i = int(b);
```

in which case false turns into 0 and true into 1. This process goes under the C/C++ name of "integral promotion".

A pointer, integer, or enumeration can be converted to a bool. A null pointer or zero value becomes false, while any other value becomes true. Such conversion is required for conditional statements:

```
char* p;

...

if (p)
    ...
```

In this example "p" is converted to bool and then the true/false value is checked to determine whether to execute the conditional block of code.
Why is a bool type an advantage? You can get a variety of opinions on whether this is a step forward. In C, common usage to mimic this type would be as follows:

```
typedef int Bool;

#define FALSE 0
#define TRUE 1
```

One problem with such an approach is that it's not at all type-safe. For example, a programmer could say:

```
Bool b;

b = 37;
```

and the compiler wouldn't care. Another problem is displaying values of Boolean type:

```
printf("%s", b ? "true" : "false");
```

which is awkward. In C++ it is possible to set up a stream I/O output operator specifically for a particular type, and thus output of bool values can be distinguished from plain integral types. This is an example of function overloading (see next section). Without bool as a distinct type, usage like:

```
void f(int i) {}

void f(Bool b) {}
```

would be invalid.
Finally, why wasn't bool added to the language, but as a class type found in a standard library? This question is hard to answer, but one possible reason is that many C implementations have supplied a Boolean pseudo-type using a typedef and #define scheme as illustrated above, and these implementations rely on representing Booleans as integral types rather than as class types.

(further comment)
In the last issue we talked about the new fundamental type "bool". Two additional comments should be made about this feature. An example of how Boolean has been faked in C was given:

```
typedef int Bool;

#define FALSE 0
#define TRUE 1
```

and then usage like this:

```
Bool b;

b = 37;
```

was presented, with a comment that a C compiler would not complain. A C++ compiler given similar usage:

```
bool b;

b = 37;
```

will not complain either, but the two sequences are not the same. In the C case, a later statement like:

```
if (b == TRUE)
    ...
```

will fail, because it reduces to:

```
if (37 == 1)
    ...
```

In the C++ case, the statement:

```
b = 37;
```

turns into:

```
b = true;
```

and a later test:

```
if (b == true)
    ...
```

will indeed succeed.

The issue was also raised as to why bool was not implemented as a class type in some C++ standard library. Dag Bruck of the ANSI/ISO C++ committees sent an example of why this will not work.

There is a rule in C++ that says that at most one user-defined conversion may be automatically applied. A user-defined conversion is a constructor like:

```
class A {
public:
    A(int);
};
```

to convert an int to an A, or a conversion function:

```
class A {
public:
    operator int();
};
```

to convert an A to an int.

If bool is a class type, for example:

```
class bool {
public:
    operator int();
};
```

then the call "f(3 < 4)" in this code:

```
class X {
public:
    X(int);
```

```
};

void f(X);

main()
{
    f(3 < 4);
}
```

will result in two user-defined conversions, one to convert the bool class object resulting from "3 < 4" to an int, the other to call the X(int) constructor on the resulting int.

Stream I/O

INTRODUCTION TO STREAM I/O PART 1 - OVERLOADING <<

In this issue we will begin discussing the stream I/O package that comes with C++. The first four sections of this issue are related and present several aspects of stream I/O along with some related topics.

If you've used C++ at all, you've probably seen a simple example of how to do output:

```
cout << "Hello, world" << "\n";
```

instead of:

```
printf("Hello, world\n");
```

cout is an output stream, kind of like stdout in C. The C example could be written as:

```
fprintf(stdout, "Hello, world\n");
```

which makes this correspondence a bit clearer.

Once you get beyond simple input/output usage, what is the stream I/O package good for? One quite useful thing it can do is to allow the programmer to take control of I/O for particular C++ types such as classes. This end is achieved by the use of operator overloading.

Suppose that we have a Date class:

```
class Date {
    int month;
    int day;
    int year;
public:
    Date(char*);
    Date(int, int, int);
};
```

with an internal representation of a Date using three integers for month, day, and year, and a couple of constructors to create a Date object. How would we output the value of a Date object?

One way would be to devise a member function:

```
void out();
```

implemented as:

```
void Date::out()
{
    printf("%d/%d/%d", month, day, year);
}
```

This function would operate on an object instance of a Date and would access the month/day/year members and display them. This approach will certainly work and may be suitable in some kinds of applications.

But this scheme doesn't integrate very well with stream I/O. For example, I cannot say:

```
Date d(9, 25, 1956);

cout << "Today's date is " << d;
```

but must say:

```
printf("Today's date is ");
d.out();
```

For this purpose it is necessary to overload the << operator. We can add a friend function to Date:

```
friend ostream& operator<<(ostream& os, const Date& d);
```

with definition:

```
ostream& operator<<(ostream& os, const Date& d)
{
    return os << d.month << "/" << d.day << "/" << d.year;
}
```

With this definition, it is possible to say:

```
Date d(9, 25, 1956);

cout << "Today's date is " << d << "\n";
```

Several aspects of this example need explanation. An overloaded operator in C++ is an operator like "+" or "<<" that is given a special meaning for certain kinds of arguments, and turns into a function. Wherever the operator is used with these arguments, a function is called. So, for example, an output statement:

```
cout << "xxx";
```

is actually:

```
cout.operator<<("xxx");
```

which is a valid function call in C++ if you're using stream I/O.

cout is an instance of class ostream (at least conceptually; the actual hierarchy is a bit complicated). When we wrote the actual statement to output a formatted Date:

```
return os << d.month << "/" << d.day << "/" << d.year;
```

we returned the ostream reference so as to allow << usage to be chained. Because << operators group left to right, a sequence like:

```
cout << "x" << "y";
```

actually means:

```
cout.operator<<("x").operator<<("y");
```

Finally, the reason that we declared operator<<(ostream&, const Date&) as a friend and not a member is that a member function that is a binary operator has an implicit convention on argument usage, namely, that for some operator @:

```
x @ y
```

means:

```
x.operator@(y);
```

that is, the left operand of the operator must be an instance of the class of which the overloaded operator is a member.

INTRODUCTION TO STREAM I/O PART 2 - FORMATTING AND MANIPULATORS

In this issue we will talk further about stream I/O. An excellent book on the subject is Steve Teale's "C++ IOStreams Handbook" (Addison-Wesley, $40). It has 350 pages with many examples and thoroughly covers I/O streams. It should be noted that I/O streams are undergoing revision as part of the ANSI/ISO standardization process. The examples we present are based on C++ headers and libraries in common use.

One obvious question about stream I/O is how to do formatting. A simple operation like:

```
int n = 37;

cout << n;
```

is equivalent to:

```
printf("%d", n);
```

that is, no special formatting is done.

But what if you want to say:

```
printf("%08d", n);
```

displaying n in a field 8 wide with leading 0s? Such an operation would be performed by saying:

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

/* stuff */

cout << setfill('0') << setw(8) << n;
```

setfill() and setw() are examples of I/O stream manipulators. A manipulator is a data object of a type known to I/O streams, that allows a user to change the state of a stream. We'll see how manipulators are implemented in a moment.

The operation illustrated here first sets the fill character to '0' and then the width of the field to 8 and then outputs the number. Some I/O stream settings like the fill character and left/right justification persist, but the width is reset after each output item.

A similar way of changing stream state is to use regular member function calls. For example,

```
cout.setf(ios::left);

cout << setfill('0') << setw(8) << n;
```

produces output:

```
37000000
```

with left justification.

In this example, you see "ios::left" mentioned. What is ios? ios is the base class of the streams class hierarchy. In the implementation used here, the hierarchy is:

```
class ios { /* stuff */ };
```

```
class ostream : public ios { /* stuff */ }

class ostream_withassign : public ostream { /* stuff */ }
```

and cout is an object instance of ostream_withassign. That is, there is a base class (ios), and the output streams class (ostream) derives or inherits from it, and ostream_withassign derives from ostream. Chapter #10 of Teale's book mentioned above discusses the rationale for the ostream_withassign class.

A statement like:

```
cout.setf(ios::left);
```

calls the member function setf() inherited from the ios class, to set flags for the stream. ios::left is an enumerator representing a particular flag value.
How can you design your own manipulators? A simple example is as follows:

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

ostream& dash(ostream& os)
{
    return os << " -- ";
}

main()
{
    cout << "xxx" << dash << "yyy" << endl;

    return 0;
}
```

We define a manipulator called "dash" that inserts a dash into an output stream. This is followed by the output of more text and then a builtin manipulator ("endl") is called. endl inserts a newline character and flushes the output buffer. We will say more about endl later in the newsletter.
Manipulators are in fact pointers to functions, and they are implemented via a couple of hooks in iostream.h:

```
ostream& operator<<(ostream& (*)(ostream&));
ostream& operator<<(ios& (*)(ios&));
```

These operators are member functions of class ostream. They will accept either a pointer to function that takes an ostream& or a pointer to function that takes an ios&. The former would be used for actual output, the latter for setting ios flags as discussed above.

INTRODUCTION TO STREAM I/O PART 3 - COPYING FILES

Suppose that you're writing a program to copy from standard input to standard output. A common way of doing this is to say:

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

```
int main(int argc, char* argv[])
{
	FILE* fpin;
	FILE* fpout;
	int c;

	assert(argc == 3);

	fpin = fopen(argv[1], "r");
	fpout = fopen(argv[2], "w");

	assert(fpin && fpout);

	while ((c = getc(fpin)) != EOF)
		putc(c, fpout);

	fclose(fpin);
	fclose(fpout);

	return 0;
}
```

EOF is a marker used to signify the end of file; its value typically is -1. In most commonly-used operating systems there is no actual character in a file to signify end of file.
This approach works on text files. Unfortunately, however, for binary files, an attempt to copy a 10406-byte file resulted in output of only 383 bytes. Why? Because EOF is itself a valid character that can occur in a binary file. If set to -1, then this is equivalent to 255 or 0377 or 0xff, a perfectly legal byte in a file. So we would need to say:

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

int main(int argc, char* argv[])
{
	FILE* fpin;
	FILE* fpout;
	int c;

	assert(argc == 3);

	fpin = fopen(argv[1], "rb");
	fpout = fopen(argv[2], "wb");

	assert(fpin && fpout);

	for (;;) {
		c = getc(fpin);
		if (feof(fpin))
			break;
```

```
            fputc(c, fpout);
        }

        fclose(fpin);
        fclose(fpout);

        return 0;
    }
```

feof() is a macro that tells whether the previous operation, in this case getc(), hit end of file. Note also that we open the files in binary mode.
How would we do the equivalent in C++? One way would be to say:

```
    #include <fstream.h>
    #include <assert.h>

    int main(int argc, char* argv[])
    {
        assert(argc == 3);

        ifstream ifs(argv[1], ios::in | ios::binary);
        ofstream ofs(argv[2], ios::out | ios::binary);
        assert(ifs && ofs);

        char c;

        while (ifs.get(c))
            ofs.put(c);

        return 0;
    }
```

ifstream and ofstream are input and output file streams, taking a single char* argument and a set of flags.
These classes are derived from ios, which has an operator conversion function (from a stream object to void*). If a statement like:

```
    assert(ifs && ofs);
```

is specified, then this conversion function is called. It returns 0 if there's something wrong with the stream. In other words, an object like "ifs" is converted to a void* automatically, and the value of the void* pointer tells the stream status (non-zero for a good state, zero for bad). The actual copying is straightforward, using the get() member function. It accepts a reference to a character, so there's no need to use the return value to pass back the character that was read.

A somewhat terser approach would be to say:

```
    #include <fstream.h>
    #include <assert.h>

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

```
{
    assert(argc == 3);

    ifstream ifs(argv[1], ios::in | ios::binary);
    ofstream ofs(argv[2], ios::out | ios::binary);
    assert(ifs && ofs);

    ofs << ifs.rdbuf();

    return 0;
}
```

with no loop involved. The expression:

```
ifs.rdbuf()
```

returns a filebuf*, a pointer to an object that actually represents the low-level buffering for the file. filebuf is derived from a class streambuf, and ofstream is derived from ostream, and ostream has an operator<< defined for streambufs. So the looping over the input file occurs within operator<<. We are "outputting" a filebuf/streambuf.

Finally, how about code for copying standard input to output:

```
#include <iostream.h>

int main()
{
    char c;

    while (cin >> c)
        cout << c;

    return 0;
}
```

If you run this program on text input, you will notice that the output's pretty jumbled. This is because by default whitespace is skipped on input. To fix this problem, you can say:

```
#include <iostream.h>

int main()
{
    char c;

    cin.unsetf(ios::skipws);

    while (cin >> c)
        cout << c;

    return 0;
}
```

to disable the skipws flag. This program does not, however, work with binary files. To make it work gets into a tricky issue; the binary mode is specified when opening a file, and in this example standard input and output are already open. This ties in with low-level buffering and

reading the first chunk of a file when it's opened. By contrast, skipping whitespace is a higher-level operation in the stream I/O library.
(correction)
In issue #008 we talked about copying files and said this about one of the examples of copying files using C:

> This approach works on text files. Unfortunately, however, for binary files, an attempt to copy a 10406-byte file resulted in output of only 383 bytes. Why? Because EOF is itself a valid character that can occur in a binary file. If set to -1, then this is equivalent to 255 or 0377 or 0xff, a perfectly legal byte in a file.

This isn't quite the case. A common mistake when copying files in C is to use a char instead of an int with getc() and putc(). If a char is used, then the explanation above is correct, because with a binary file EOF interpreted as a character is one of the 256 valid bit patterns that a char can hold.
But with an int this is not a problem. getc(), and its functional equivalent fgetc(), return an unsigned char converted to an int. So the int can represent all character values 0-255, along with the EOF marker (typically -1).
It turns out that the reason why the example failed was due to a ^Z in the file. ^Z used to be used as an end-of-file marker for DOS files used on PCs.
Thanks to David Nelson for mentioning this.

INTRODUCTION TO STREAM I/O PART 4 - TIE()

In issue #008 we talked about copying files using a variety of methods. One example that was presented was this one:

```
#include <iostream.h>

int main()
{
        char c;

        cin.unsetf(ios::skipws);

        while (cin >> c)
                cout << c;

        return 0;
}
```

Jerry Schwarz suggested that it might be worth discussing the tie() function and its effect on the performance of this code. Specifically, if we slightly change the above code to:

```
#include <iostream.h>

int main()
{
        char c;

        cin.tie(0);
```

```
        cin.unsetf(ios::skipws);

        while (cin >> c)
            cout << c;

        return 0;
    }
```

it runs about 8X faster with one popular C++ compiler, and about 18X with another.
The difference has to do with buffering and flushing of streams. When input is requested, for example with:

```
    cin >> c
```

there may be output pending in the buffer for the output stream. The input stream is therefore tied to the output stream such that a request for input will cause pending output to be flushed. Flushing output is expensive, typically triggering a flush() call and a write() system call (on UNIX systems). Disabling the linkage between the input and output streams gets rid of this overhead.
To further illustrate this point, consider another example:

```
    #include <iostream.h>

    int main()
    {
        char buf[100];

        //cin.tie(0);

        cin.unsetf(ios::skipws);

        cout.unsetf(ios::unitbuf);

        cout << "What is your name? ";

        cin >> buf;

        return 0;
    }
```

It's common for output to be completely unbuffered (unit buffering) if going to a terminal (screen or window). So setting cin.tie(0) will not necessarily change observable behavior, because output will be flushed immediately in all cases.
To affect behavior in this example, one also needs to disable unit buffering for the stream, achieved by saying:

```
    cout.unsetf(ios::unitbuf);
```

Once this is done, cin.tie(0) will change behavior in a visible way. If the input stream is untied, then the prompt in the example above will not come out before input is requested from the user, leading to confusion.
Note also that current libraries vary in their behavior. The above example works for one library that was tried, but for another, there appears to be no way to disable unit buffering

under any circumstances, when output is to a terminal. The draft ANSI/ISO C++ standard calls for unit buffering to be set for error output ("cerr").
If tie() is called with no argument, it returns the stream currently tied to. For example:

```
cout << (void*)cin.tie() << "\n";

cout << (void*)(&cout) << "\n";
```

give identical results if cin is currently tied to cout.
Copying files a character at a time has other pitfalls. One has to be careful in assessing the buffering and function call overhead for anything done on a per-character basis. There is yet another way of copying files by character, using streambufs, that we'll present in a future issue.

INTRODUCTION TO STREAM I/O PART 5 - STREAMBUF

In previous issues we talked about various ways of copying files using stream I/O, some of the ways of affecting I/O operations by specifying unit buffering or not and tying one stream to another, and so on.
Another way of copying input to output using stream I/O is to say:

```
#include <iostream.h>

int main()
{
    int c;

    while ((c = cin.rdbuf()->sbumpc()) != EOF)
        cout.rdbuf()->sputc(c);

    return 0;
}
```

This scheme uses what are known as streambufs, underlying buffers used in the stream I/O package. An expression:

```
cin.rdbuf()->sbumpc()
```

says "obtain the streambuf pointer for the standard input stream (cin) and grab the next character from it and then advance the internal pointer within the buffer". Similarly,

```
cout.rdbuf()->sputc(c)
```

adds a character to the output buffer.
Doing I/O in this way is lower-level than some other approaches, but correspondingly faster. If we summarize the four file-copying methods we've studied (see issues #008 and #009 for code examples of them), from slowest to fastest, they might be as follows.
Copy a character at a time with >> and <<:

```
cin.tie(0);
cin.unsetf(ios::skipws);

while (cin >> c)
    cout << c;
```

Copy using get() and put():

```
ifstream ifs(argv[1], ios::in | ios::binary);
```

```
    ofstream ofs(argv[2], ios::out | ios::binary);

    while (ifs.get(c))
        ofs.put(c);
```

Copy with streambufs (above):

```
    while ((c = cin.rdbuf()->sbumpc()) != EOF)
        cout.rdbuf()->sputc(c);
```

Copy with streambufs but explicit copying buried:

```
    ifstream ifs(argv[1], ios::in | ios::binary);
    ofstream ofs(argv[2], ios::out | ios::binary);

    ofs << ifs.rdbuf();
```

A table of relative times, for one popular C++ compiler, comes out like so:

>>, <<	100
get/put	72
streambuf	62
streambuf hidden	43

Actual times will vary for a given library. Perhaps the most critical factor is whether functions that are used in a given case are inlined or not. Note also that if you are copying binary files you need to be careful with the way copying is done.

Why the time differences? All of these methods use streambufs in some form. But the slowest method, using >> and <<, also does additional processing. For example, it calls internal functions like ipfx() and opfx() to handle unit buffering, elision of whitespace on input, and so on. get/put also call these functions.

The fastest two approaches do not worry about such processing, but simply allow one to manipulate the underlying buffer directly. They offer fewer services but are correspondingly faster.

INTRODUCTION TO STREAM I/O PART 6 - SEEKING IN FILES

In earlier issues we talked about streambufs, the underlying buffer used in I/O operations. One of the things that you can do with a buffer is position its pointer at various places. For example:

```
    #include <fstream.h>

    int main()
    {
        ofstream ofs("xxx");
        if (!ofs)
            ; // give error

        ofs << ' ';

        ofs << "abc";
```

```
        streampos pos = ofs.tellp();
        ofs.seekp(0);
        ofs << 'x';
        ofs.seekp(pos);

        ofs << "def\n";

        return 0;
    }
```

Here we have an output file stream attached to a file "xxx". We open this file and write a single blank character at the beginning of it. In this particular application this character is a status character of some sort that we will update from time to time.

After writing the status character, we write some characters to the file, at which point we wish to update the status character. To do this, we save the current position of the file using tellp(), seek to the beginning, write the character, and then seek back to where we were, at which point we can write some more characters.

Note that "streampos" is a defined type of some kind rather than simply a fixed fundamental type like "long". You should not assume particular types when working with file offsets and positions, but instead save the value returned by tellp() and then use it later.

In a similar way, it's tricky to use absolute file offsets other than 0 when seeking in files. For example, there are issues with binary files and with CR/LF translation. You may be assuming that a newline takes two characters when it only takes one, or vice versa.

seekp() also has a two-parameter version:

```
ofs.seekp(pos, ios::beg);  // from beginning

ofs.seekp(pos, ios::cur);  // from current position

ofs.seekp(pos, ios::end);  // from end
```

As we've said before, this area is in a state of flux, pending standardization. So future usage may be somewhat different than shown here.

C++ Virtual Functions

Imagine that you are doing some graphics programming, with a variety of shapes to be output to the screen. Initially, you want to support Line, Circle, and Text. Each shape has an X,Y origin and a color.

How might this be done in C++? One way is to use virtual functions. A virtual function is a function member of a class, declared using the "virtual" keyword. A pointer to a derived class object may be assigned to a base class pointer, and a virtual function called through the pointer. If the function is virtual and occurs both in the base class and in derived classes, then the right function will be picked up based on what the base class pointer "really" points at. For graphics, we can use a base class called Shape, with derived classes named Line, Circle, and Text. Shape and each of the derived classes has a virtual function draw(). We create new objects and point at them using Shape* pointers. But when we call a draw() function, as in:

```
Shape* p = new Line(0.1, 0.1, Co_blue, 0.4, 0.4);

p->draw();
```

the draw() function for a Line is called, not the draw() function for Shape. This style of programming is very common and goes by names like "polymorphism" and "object-oriented programming". To illustrate it further, here is an example of this type of programming for a graphics application. Annotations in /* */ explain in some detail what is going on.

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

typedef double Coord;

/*
The type of X/Y points on the screen.
*/

enum Color {Co_red, Co_green, Co_blue};

/*
Colors.
*/

// abstract base class for all shape types
class Shape {
protected:
    Coord xorig; // X origin
    Coord yorig; // Y origin
    Color co; // color

/*
These are protected so that they can be accessed
by derived classes.  Private wouldn't allow this.

These data members are common to all shape types.
*/

public:
    Shape(Coord x, Coord y, Color c) :
```

```
        xorig(x), yorig(y), co(c) {} // constructor

/*
Constructor to initialize data members common to
all shape types.
*/

    virtual ~Shape() {} // virtual destructor

/*
Destructor for Shape.  It's a virtual function.
Destructors in derived classes are virtual also
because this one is declared so.
*/

    virtual void draw() = 0; // pure virtual draw() function

/*
Similarly for the draw() function.  It's a pure virtual and
is not called directly.
*/

};

// line with X,Y destination
class Line : public Shape {

/*
Line is derived from Shape, and picks up its
data members.
*/

    Coord xdest; // X destination
    Coord ydest; // Y destination

/*
Additional data members needed only for Lines.
*/

public:
    Line(Coord x, Coord y, Color c, Coord xd, Coord yd) :
      xdest(xd), ydest(yd),
      Shape(x, y, c) {} // constructor with base initialization

/*
Construct a Line, calling the Shape constructor as well
to initialize data members of the base class.
```

```
*/

        ~Line() {cout << "~Line\n";} // virtual destructor

/*
Destructor.
*/

        void draw() // virtual draw function
        {
                cout << "Line" << "(";
                cout << xorig << ", " << yorig << ", " << int(co);
                cout << ", " << xdest << ", " << ydest;
                cout << ")\n";
        }

/*
Draw a line.
*/

};

// circle with radius
class Circle : public Shape {
        Coord rad; // radius of circle

/*
Radius of circle.
*/

public:
        Circle(Coord x, Coord y, Color c, Coord r) : rad(r),
           Shape(x, y, c) {} // constructor with base initialization

        ~Circle() {cout << "~Circle\n";} // virtual destructor
        void draw() // virtual draw function
        {
                cout << "Circle" << "(";
                cout << xorig << ", " << yorig << ", " << int(co);
                cout << ", " << rad;
                cout << ")\n";
        }
};

// text with characters given
class Text : public Shape {
        char* str; // copy of string
```

```
public:
    Text(Coord x, Coord y, Color c, const char* s) :
      Shape(x, y, c) // constructor with base initialization
    {
        str = new char[strlen(s) + 1];
        assert(str);
        strcpy(str, s);

/*
Copy out text string.  Note that this would be done differently
if we were taking advantage of some newer C++ features like
exceptions and strings.
*/

    }
    ~Text() {delete [] str; cout << "~Text\n";} // virtual dtor

/*
Destructor; delete text string.
*/

    void draw() // virtual draw function
    {
        cout << "Text" << "(";
        cout << xorig << ", " << yorig << ", " << int(co);
        cout << ", " << str;
        cout << ")\n";
    }
};

int main()
{
    const int N = 5;
    int i;
    Shape* sptrs[N];

/*
Pointer to vector of Shape* pointers.  Pointers to classes
derived from Shape can be assigned to Shape* pointers.
*/

    // initialize set of Shape object pointers

    sptrs[0] = new Line(0.1, 0.1, Co_blue, 0.4, 0.5);
    sptrs[1] = new Line(0.3, 0.2, Co_red, 0.9, 0.75);
    sptrs[2] = new Circle(0.5, 0.5, Co_green, 0.3);
    sptrs[3] = new Text(0.7, 0.4, Co_blue, "Howdy!");
```

```
        sptrs[4] = new Circle(0.3, 0.3, Co_red, 0.1);

    /*
    Create some shape objects.
    */

        // draw set of shape objects

        for (i = 0; i < N; i++)
            sptrs[i]->draw();

    /*
    Draw them using virtual functions to pick up the
    right draw() function based on the actual object
    type being pointed at.
    */

        // cleanup

        for (i = 0; i < N; i++)
            delete sptrs[i];

    /*
    Clean up the objects using virtual destructors.
    */

        return 0;
    }
```

When we run this program, the output is:

```
    Line(0.1, 0.1, 2, 0.4, 0.5)
    Line(0.3, 0.2, 0, 0.9, 0.75)
    Circle(0.5, 0.5, 1, 0.3)
    Text(0.7, 0.4, 2, Howdy!)
    Circle(0.3, 0.3, 0, 0.1)
    ~Line
    ~Line
    ~Circle
    ~Text
    ~Circle
```

with enum color values represented by small integers.

A few additional comments. Virtual functions typically are implemented by placing a pointer to a jump table in each object instance. This table pointer represents the "real" type of the object, even though the object is being manipulated through a base class pointer.

Because virtual functions usually need to have their function address taken, to store in a table, declaring them inline as the above example does is often a waste of time. They will be laid down as static copies per object file. There are some advanced techniques for optimizing virtual functions, but you can't count on these being available.

Note that we declared the Shape destructor virtual (there are no virtual constructors). If we had not done this, then when we iterated over the vector of Shape* pointers, deleting each object in turn, the destructors for the actual object types derived from Shape would not have been called, and in the case above this would result in a memory leak in the Text class.
Shape is an example of an abstract class, whose purpose is to serve as a base for derived classes that actually do the work. It is not possible to create an actual object instance of Shape, because it contains at least one pure virtual function.

Templates

INTRODUCTION TO C++ TEMPLATES PART 1 - FUNCTION TEMPLATES

In issue #007 we talked about the use of inline functions. Suppose that you wish to compute the maximum of two quantities, and you define a C macro for this:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

This works OK until a case like:

```
max(x++, y++)
```

comes along. An inline function:

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}
```

solves this problem. But what if you want a max() function for a variety of numeric types? You can define a slew of function prototypes:

```
int max(int, int);

long max(long, long);

double max(double, double);
```

and rely on function overloading to sort things out. Or you can define one function that might work in all cases:

```
long double max(long double, long double);
```

since nothing can be bigger than a long double, right? This last approach fails because there's no guarantee that, for example, the size of a long is less than the size of a long double, and assigning a long to a long double would in such a case result in loss of precision.
In C++ there is another way to approach this problem, using what are called parameterized types or templates. We can define a function template:

```
template <class T> inline T max(T a, T b)
{
    return a > b ? a : b;
}
```

The preface "template <class T>" is used to declare a template. T is a template parameter, a type argument to the template. When this template is used:

```
int a = 37;
int b = 47;

int i = max(a, b);
```

the type value of T will be "int", because a and b are integers. If instead I had said:

```
double a = 37.53;
double b = -47.91;

double d = max(a, b);
```

then T would have the type value "double". The process of generating an actual function from a function template is known as "instantiation". Note also that "const T&" may be used instead of "T"; we will be discussing this point in a future issue of the newsletter.

This template will also work on non-numeric types, so long as they have the ">" operator defined. For example:

```
class A {
public:
    int operator>(const A&); // use "bool" return type
                             // instead, if available
};

A a;
A b;

A c = max(a, b);
```

Templates are a powerful but complex feature, about which we will have more to say. Languages like C or Java(tm), that do not have templates, typically use macros or rely on using base class pointers and virtual functions to synthesize some of the properties of templates.

Templates in C++ are a more ambitious attempt to support "generic programming" than some previous efforts found in other programming languages. Support for generic programming in C++ is considered by some to be as important a language goal for C++ as is support for object-oriented programming (using base/derived classes and virtual functions; see newsletter issue #008). An example of heavy template use can be found in STL, the Standard Template Library.

NEW C++ FEATURE - MEMBER TEMPLATES

In chapter 14 of the draft ANSI/ISO C++ standard is a mention of something called member templates. This feature is new in a way and not yet widely available, but worth mentioning here.

Member templates are simply a generalization of templates such that a template can be a class member. For example:

```
#include <stdio.h>

template <class A, class B> struct Pair {
    A a;
    B b;

    Pair(const A& ax, const B& bx) : a(ax), b(bx) {}

    template <class T, class U> Pair(const Pair<T,U>& p) :
      a(p.a), b(p.b) {}
};

int main()
{
    Pair<short, float> x(37, 12.34);
    Pair<long, long double> y(x);
```

```
        printf("%ld %Lg\n", y.a, y.b);

        return 0;
    }
```

This is an adaptation of a class found in the Standard Template Library. Note that an object of class Pair<long, long double> is constructed from an object of class Pair<short, float>. By using a template constructor it is possible to construct a Pair from any other Pair, assuming that conversion from T to A and U to B are supported. Without the availability of template constructors one could only declare constructors with fixed types like "Pair(int)" or else use the template arguments to Pair itself, as in "Pair(A, B)".
In a similar way to function template use, it's possible to have usage like:

```
    template <class T> struct A {
        template <class U> struct B {/* stuff */};
    };

    A<double>::B<long> ab;
```

In this example, the type value of T within the nested template declaration would be "double", while the value of U would be "long".
There are a few restrictions on member templates. A destructor for a class cannot be defined as a function template, nor may a function template member of a class be virtual.

INTRODUCTION TO C++ TEMPLATES PART 2 - CLASS TEMPLATES

To continue our introduction of C++ templates, we'll be saying a few things about class templates in this issue. Templates are a part of the language still undergoing major changes, and it's tricky to figure out just what to say. But we'll cover some basics that are well accepted and in current usage.
A skeleton for a class template, and definitions of a member function and a static data item, looks like this:

```
    template <class T> class A {
        void f();
        static T x;
    };

    template <class T> void A<T>::f()
    {
        // stuff
    }

    template <class T> T A<T>::x = 0;
```

T is a placeholder for a template type argument, and is bound to that argument when the template is instantiated. For example, if I say:

```
    A<double> a;
```

then the type value of T is "double". The binding of template arguments and the generation of an actual class from a template is a process known as "instantiation". You can view a template as a skeleton or macro or framework. When specific types, such as double, are added to this skeleton, the result is an actual C++ class.

Template arguments may also be constant expressions:

```
template <int N> struct A {
    // stuff
};

A<-37> a;
```

This feature is useful in the case where you want to pass a size into the template. For example, a Vector template might accept a type argument that tells what type of elements will be operated on, and a size argument giving the vector length:

```
template <class T, int N> class Vector {
    // stuff
};

Vector<float, 100> v;
```

A template argument may have a default specified (this feature is not widely available as yet):

```
template <class T = int, int N = 100> class Vector {
    // stuff
};

Vector<float, 50> v1;          // Vector<float, 50>
Vector<char> v2;               // Vector<char, 100>
Vector<> v3;                   // Vector<int, 100>
```

To see how some of these basic ideas fit together, let's actually build a simple Vector template, with set() and get() functions:

```
template <class T, int N = 100> class Vector {
    T vec[N];
public:
    void set(int pos, T val);
    T get(int pos);
};

template <class T, int N> void Vector<T, N>::set(int pos, T val)
{
    if (pos < 0 || pos >= N)
        ; // give error of some kind

    vec[pos] = val;
}

template <class T, int N> T Vector<T, N>::get(int pos)
{
    if (pos < 0 || pos >= N)
        ; // give error of some kind

    return vec[pos];
}
```

```
// driver program

int main()
{
	Vector<double, 10> v;
	int i = 0;
	double d = 0.0;

	// set locations in vector

	for (i = 0; i < 10; i++)
		v.set(i, double(i * i));

	// get location values from vector

	for (i = 0; i < 10; i++)
		d = v.get(i);

	return 0;
}
```

Actual values are stored in a private vector of type T and length N. In a real Vector class we might overload operator[] to provide a natural sort of interface such as an actual vector has. What would happen if we said something like:

```
Vector<char, -1000> v;
```

This is an example of code that is legal until the template is actually instantiated into a class. Because a member like:

```
char vec[-1000];
```

is not valid (you can't have arrays of negative or zero size), this usage will be flagged as an error when instantiation is done.

The process of instantiation itself is a bit tricky. If I have 10 translation units (source files), and each uses an instantiated class:

```
Vector<unsigned long, 250>
```

where does the code for the instantiated class's member functions go? The template definition itself resides most commonly in a header file, so that it can be accessed everywhere and because template code has some different properties than other source code.

This is an extremely hard problem for a compiler to solve. One solution is to make all template functions inline and duplicate the code for them per translation unit. This results in very fast but potentially bulky code.

Another approach, which works if you have control over the object file format and the linker, is to generate duplicate instantiations per object file and then use the linker to merge them.

Yet another approach is to create auxiliary files or directories ("repositories") that have a memory of what has been instantiated in which object file, and use that state file in conjunction with the compiler and linker to control the instantiation process.

There are also schemes for explicitly forcing instantiation to take place. We'll discuss these in a future issue. The instantiation issue is usually hidden from a programmer, but sometimes becomes visible, for example if the programmer notices that object file sizes seem bloated.

INTRODUCTION TO TEMPLATES PART 3 - TEMPLATE ARGUMENTS

In previous issues we talked about setting up a class template:

```
template <class T> class A {
    T x;
    // stuff
};
```

When this template is used:

```
A<double> a;
```

the type "double" gets bound to the formal type parameter T, part of a process known as instantiation. In this example, the instantiated class will have a data member "x" of type double.

What sorts of arguments can be used with templates? Type arguments are allowed, including "void":

```
template <class T> class A {
    T* x;
};

A<void> a;
```

The member x will be of type void* in this case. In the earlier example, using void as a type argument would result in an instantiation error, because a data member of a class (or any object for that matter) cannot be of type void.

Usage like:

```
A<int [37][47]> a1;

A< A<void> > a2;
```

is also valid. Note that in the second example, the second space is required, because ">>" is an operator meaning right shift.

It's also possible to have non-type arguments. Constant expressions can be used:

```
template <class T, int n> class A {
    T vec[n];
};

A<float, 100> a;
```

This is useful in specifying the size of an internal data structure, in this example a vector of float[100]. The size could also be specified via a constructor, but in that case the size would not be known at compile time and therefore dynamic storage would have to be used to allocate the vector.

The address of an external object can be used:

```
template <char* cp> struct A { /* ... */ };

char c;

A<&c> a;
```

or you can use the address of a function:

```
template <void (*fp)(int)> struct A { /* ... */ };
```

```
void f(int) {}

A<f> a;
```

This latter case might be useful if you want to pass in a pointer of a function to be used internally within the template, for example, to compare elements of a vector.
Some other kinds of constructs are not permitted as arguments:

- a constant expression of floating type

- addresses of array elements

- addresses of non-static class members

- local types

- addresses of local objects

Two template classes (a template class is an instantiated class template, that is, a template with specific arguments) refer to the same class if their template names are identical and in the same scope and their arguments have identical values. For example:

```
template <class T, int n> struct A {};

typedef short* TT;

A<short*, 100> a1;
A<TT, 25*4> a2;
```

a1 and a2 are of the same type.

INTRODUCTION TO TEMPLATES PART 4 - SPECIALIZATIONS

In previous issues we've covered some of the basics of C++ templates. Recall that a template is a class or function skeleton, and is combined with specified type arguments to produce an actual class or function.
Beyond this general mechanism, C++ also allows the programmer to define specialized classes and functions that take the template and implement it for particular types of template arguments.
Suppose, for example, that you have a String template, that supports strings of most anything -- chars, ints, doubles, arbitrary class types, and so on. Now, it's pretty likely that strings of characters will be used heavily, so it might make sense to special case this combination of template and template argument:

```
template <class T> class String {
    // stuff
};

template <> class String<char> {
    // stuff
};

String<char> x;
```

The "template <>" notation is fairly new and may not yet be implemented in your local compiler.
This sequence is a bit different from:

```
template <class T> class String {
    // stuff
};

String<char> x;
```

In this second case, the default implementation of String is used, whereas in the specialization case, the programmer overrides the default template and provides an implementation of String<char>.
For a function template, a specialization would be defined as:

```
template <class T> void f(T) {}

template <> void f(int) {}
```

It's also possible to have forward declarations of specializations:

```
template <class T> class String {};

template <> class String<double>;
```

or:

```
template <class T> void f(T) {}

template <> void f(unsigned short&);
```

A specialization must be declared or defined before use, so for example:

```
template <class T> int f(T) {return 0;}

int i = f(12.34);

template <> int f(double) {return 37;}
```

is invalid.
An interesting quirk with function templates concerns the case where you have a C function, mixed in with a function template and specialization:

```
extern "C" void f(int);

template <class T> int f(T) {return 0;}

template <> int f(int)
{
    f(37);
    return 0;
}
```

The f(37) call here is not recursive. Both "void f(int)" and "int f(int)" match the call, and the non-template is preferred in such a case.
It's also possible to have nested specializations, as in:

```
template <class T> class A {
    template <class U> class B {
        template <class V> void f(V) {}
```

```
        };
    };

    template <> template <> template <>
        void A<float>::B<double>::f(long double) {}
```

and you can specialize special member types such as constructors:

```
    struct A {
        template <class T> A(T) {}
    };

    template <> A::A(short) {}
```

or static data members:

```
    template <class T> struct A {
        template <class U> struct B {
            static int x;
        };
    };

    template <> template <> int A<float>::B<long double>::x = 59;
```

Specializations are a way of special-casing templates for particular argument types, and are useful in a variety of applications. But they can also be abused and make code harder to understand, especially if a reader of the code doesn't pick up on the fact that specializations are present.

INTRODUCTION TO TEMPLATES PART 5 - FORCING INSTANTIATION

In previous issues we've talked about the process of template instantiation, in which template parameters are bound to actual type arguments. For example:

```
    template <class T> class A {};

    A<int> a;
```

At instantiation time, the template formal parameter T is assigned the type value "int". Instantiation is done based on need -- the generated class A<int> will not be instantiated unless it has first been referenced or otherwise used.

The actual process of instantiation is done in various ways, for example during the link phase of producing an executable program. But it is possible to explicitly force instantiation to occur in a file. For example:

```
    template <class T> class A {
        T x;
        void f();
    };
    template <class T> void A<T>::f() {}

    template class A<double>;
```

will force the instantiation of A<double>.

The whole area of instantiation is still in a state of flux, and this feature may not be available with your compiler.

INTRODUCTION TO TEMPLATES PART 6 - FRIENDS

In earlier issues we've seen how a template is something like a class, except that it can be parameterized, that is, type arguments can be supplied to create an actual class from a template through the process of instantiation.
In C++ friends are used to give outside functions and classes access to private members of a class. Friends can also be used with templates, in a similar way. For example:

```
template <class T> class A {
    int x;
    friend void f();
};

void f()
{
    A<double> a;

    int i = a.x;
}

int main()
{
    f();

    return 0;
}
```

In this example, the function f() gains access to the private members of all instantiated classes that come from the template A, such as A<double>, A<char**>, and so on.
In a similar way, a whole class can be granted access to private members of a template:

```
template <class T> class A {
    int x;
    friend class B;
};

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

void B::f()
{
    A<short> a;

    int i = a.x;
}

int main()
{
```

```
        B b;

        b.f();

        return 0;
    }
```

Here, class B is a friend of template A, and so all of B's members can access the private members of A<short>.
In an earlier issue, we talked about member templates. With this feature additional combinations of friends and templates are possible.

Use of Static

THE MEANING OF "STATIC"

Someone asked about the meaning of the term "static" in C++. This particular term is perhaps the most overworked one in the language. It's both a descriptive word and a C++ keyword that is used in various ways.
"static" as a descriptive term refers to the lifetime of C++ memory or storage locations. There are several types of storage:

- static

- dynamic (heap)

- auto (stack)

A typical storage layout scheme will have the following arrangement, from lowest to highest virtual memory address:

text (program code)

static (initialized and uninitialized data)

heap

(large virtual address space gap)

```
stack
```

with the heap and stack growing toward each other. The C++ draft standard does not mandate this arrangement, and this example is only an illustration of one way of doing it.
Static storage thus refers to memory locations that persist for the life of the program; global variables are static. Stack storage comes and goes as functions are called ("stack frames"), and heap storage is allocated and deallocated using operators new and delete. Note that usage like:

```
void f()
{
    static int x = 37;
}
```

also refers to storage that persists throughout the program, even though x cannot be used outside of f() to refer to that storage.
So we might say that "static" as a descriptive term is used to describe the lifetime of memory locations. static can also be used to describe the visibility of objects. For example:

```
static int x = 37;

static void f() {}
```

says that x and f() are not visible outside the source file where they're defined, and

```
void f()
{
    static int x = 47;
}
```

says that x is not visible outside of f(). Visibility and lifetime are not the same thing; an object can exist without being visible.
So far we've covered the uses of static that are found in C. C++ adds a couple of additional twists. It is possible to have static members of a C++ class:

```
class A {
public:
    static int x;
    static void f();
    int y;
};

int A::x = 0;

void A::f() {}
```

A static data member like A::x is shared across all object instances of A. That is, if I define two object instances:

```
A a1;
A a2;
```

then they have the same x but y is different between them. A static data member is useful to share information between object instances. For example, in issue #010 we talked about using a specialized allocator on a per-class basis to allocate memory for object instances, and a static member "freelist" was used as part of the implementation of this scheme.
A static function member, such as A::f(), can be used to provide utility functions to a class. For example, with a class representing calendar dates, a function that tells whether a given

year is a leap year might best be represented as a static function. The function is related to the operation of the class but doesn't operate on particular object instances (actual calendar dates) of the class. Such a function could be made global, but it's cleaner to have the function as part of the Date package:

```
class Date {
    static int is_leap(int year);   // use bool if available
public:
    // stuff
};
```

In this example, is_leap() is private to Date and can only be used within member functions of Date, instead of by the whole program.

static meaning "local to a file" has been devalued somewhat by the introduction of C++ namespaces; the draft standard states that use of static is deprecated for objects in namespace scope. For example, saying:

```
static int x;
static void f() {}
```

is equivalent to:

```
namespace {
    int x;
    void f() {}
}
```

That is, an unnamed namespace is used to wrap the static declarations. All unnamed namespaces in a single source file (translation unit) are part of the same namespace and differ from similar namespaces in other translation units.

LOCAL STATICS AND CONSTRUCTORS/DESTRUCTORS

There's one additional interesting angle on the use of static. Suppose that you have:

```
class A {
public:
    A();
    ~A();
};

void f()
{
    static A a;
}
```

This object has a constructor that must be called at some point. But we can't call the constructor each time that f() is called, because the object is static, that is, exists for the life of the program, and should be constructed only once. The draft standard says that such an object should be constructed once, the first time execution passes through its declaration.

This might be implemented internally by a compiler as:

```
void f()
{
    static int __first = 1;
    static A a;
```

```
        if (__first) {
            a.A::A();       // conceptual, not legal syntax
            __first = 0;
        }

        // other processing
    }
```

If f() is never called, then the object will not be constructed. If it is constructed, it must be destructed when the program terminates.

Mutable

In C++ it's possible to have a class object instance that is constant and cannot be modified by the program, once initially set up. For example:

```
    class A {
    public:
        int x;
        A();
    };

    const A a;

    void g()
    {
        a.x = 37;
    }
```

is illegal. In a similar way, invoking a non-const member function on a const object is also illegal:

```
    class A {
    public:
        int x;
        A();
        void f();
    };

    const A a;

    void g()
    {
        a.f();
    }
```

The reason for this latter prohibition is due to separate compilation. A::f() may be defined in some other translation unit, and there's no way of knowing whether it modifies the object upon which it operates.
It is possible to define const member functions:

```
void f() const;
```

that are allowed to operate on a const object instance. Such a function does not modify the instance it operates on. The type of the "this" pointer for a class T is normally:

```
T *const this;
```

meaning that the pointer cannot be changed. Within a const member function, the type is:

```
const T *const this;
```

meaning that neither the pointer nor the pointed-at object instance can be modified.
Recently a new feature has been added to C++ to selectively allow for individual data class members to be modified even for a const object instance, and lessen the need for casting away of const. For example:

```
class A {
public:
    mutable int x;
    A();
};

const A a;

void f()
{
    a.x = 37;
}
```

This says that "x" can be modified even though it's a member of a const object instance. How useful "mutable" turns out to be remains to be seen. One cited example for its use is within classes whose object instances appear constant but actually do change their state internally. For example:

```
class Box {
    double xll, yll;        // lower left X,Y
    double xur, yur;        // upper right X,Y
    double a;               // cached area
public:
    double area() const
    {
        a = (xur - xll) * (yur - yll);
        return a;
    }
    class Box(double x1, double y1, double x2, double y2) :
       xll(x1), yll(y1), xur(x2), yur(y2)
    {
    }
};

const Box b(1.0, 1.0, 11.0, 14.0);
```

```
void f()
{
    b.area();
}
```

which is illegal usage unless we instead say:

```
class Box {
    double xll, yll;        // lower left X,Y
    double xur, yur;        // upper right X,Y
    mutable double a;       // cached area
public:
    double area() const
    {
        a = (xur - xll) * (yur - yll);
        return a;
    }
    class Box(double x1, double y1, double x2, double y2) :
       xll(x1), yll(y1), xur(x2), yur(y2)
    {
    }
};

const Box b(1.0, 1.0, 11.0, 14.0);

void f()
{
    b.area();
}
```

Explicit

In C++ it is possible to declare constructors for a class, taking a single parameter, and use those constructors for doing type conversion. For example:

```
class A {
public:
    A(int);
};

void f(A) {}

void g()
```

```
{
    A a1 = 37;

    A a2 = A(47);

    A a3(57);

    a1 = 67;

    f(77);
}
```

A declaration like:

```
A a1 = 37;
```

says to call the A(int) constructor to create an A object from the integer value. Such a constructor is called a "converting constructor".
However, this type of implicit conversion can be confusing, and there is a way of disabling it, using a new keyword "explicit" in the constructor declaration:

```
class A {
public:
    explicit A(int);
};

void f(A) {}

void g()
{
    A a1 = 37;      // illegal

    A a2 = A(47);   // OK

    A a3(57);       // OK

    a1 = 67;        // illegal

    f(77);          // illegal
}
```

Using the explicit keyword, a constructor is declared to be "nonconverting", and explicit constructor syntax is required:

```
class A {
public:
    explicit A(int);
};

void f(A) {}
```

```
void g()
{
    A a1 = A(37);

    A a2 = A(47);

    A a3(57);

    a1 = A(67);

    f(A(77));
}
```

Note that an expression such as:

```
A(47)
```

is closely related to function-style casts supported by C++. For example:

```
double d = 12.34;

int i = int(d);
```

Standard Template Library

INTRODUCTION TO STL PART 1 - GETTING STARTED

STL stands for Standard Template Library, and is a new feature of C++. We will be presenting some of the basic features of STL in this and subsequent issues. STL may not be available with your local C++ compiler as yet. The examples presented here were developed with Borland C++ 5.0. Third-party versions of STL are available from companies like ObjectSpace and Rogue Wave, and HP's original implementation (which may be obsolete) is available free on the Internet.

To get an idea of the flavor of STL, let's consider a simple example, one where we wish to create a set of integers and then shuffle them into random order:

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v;
```

```
        for (int i = 0; i < 25; i++)
            v.push_back(i);

        random_shuffle(v.begin(), v.end());

        for (int j = 0; j < 25; j++)
            cout << v[j] << " ";
        cout << endl;

        return 0;
    }
```

When run, this program produces output like:

```
    6 11 9 23 18 12 17 24 20 15 4 22 10 5 1 19 13 3 14 16 0 8 21 2 7
```

There's quite a bit to say about this example. In the first place,
STL is divided into three logical parts:

- containers

- iterators

- algorithms

Containers are data structures such as vectors. They are implemented as templates, meaning that a container can hold any type of data element. In the example above, we have "vector<int>", or a vector of integers.
Iterators can be viewed as pointers to elements within a container.
Algorithms are functions (function templates actually) that operate on data in containers. Algorithms have no special knowledge of the types of data on which they operate, meaning that an algorithm is generic in its application.
We include header files for the STL features that we want to use. Note that the headers have no ".h" on them. This is a new feature in which the .h for standard headers is dropped.
The next line of interest is:

```
    using namespace std;
```

We discussed namespaces in earlier newsletter issues. This statement means that the names in namespace "std" should be made available to the program. Standard libraries use std to avoid the problem mentioned earlier where library elements (like functions or class names) conflict with names found in other libraries.
The line:

```
    vector<int> v;
```

declares a vector of integers, and then:

```
    for (int i = 0; i < 25; i++)
        v.push_back(i);
```

adds the numbers 0-24 to the vector, using the push_back() member function.
Actual shuffling is done with the line:

```
    random_shuffle(v.begin(), v.end());
```

where v.begin() and v.end() are iterator arguments that delimit the extend of the list to be shuffled.
Finally, we display the shuffled list of integers, using an overloaded operator[] on the vector:

```
    for (int j = 0; j < 25; j++)
        cout << v[j] << " ";
    cout << endl;
```

This code is quite generic. For example, we could change:

```
    vector<int> v;
```

to:

```
    vector<float> v;
```

and fill the vector with floating-point numbers. The rest of the code that shuffles and displays the result would not change.

One point to note about STL performance. The library, at least the version used for these examples, is implemented as a set of header files and inline functions (templates). This structure is probably necessary for performance, due to the internal use of various helper functions (for example, begin() in the above example). Such an architecture is very fast but can cause code size blowups in some cases.

We will be saying more about STL in future issues. The library is not yet in widespread use, and it's too early to say how it will shake out.

INTRODUCTION TO STL PART 2 - VECTORS, LISTS, DEQUES

In the previous issue we introduced the C++ Standard Template Library. STL is a combination of containers used to store data, iterators on those containers, and algorithms to manipulate containers of data. STL uses templates and inline functions very heavily.

In this issue we'll talk about some of the types of containers that are available for holding data, namely vectors, lists, and deques.

A vector is like a smart array. You can use [] to efficiently access any element in the vector, and the vector grows on demand. But inserting into the middle of a vector is expensive, because elements must be moved down, and growing the vector is costly because it must be copied to another vector internally.

A list is like a doubly-linked list that you've used before. Insertion or splicing of subsequences is very efficient at any point in the list, and the list doesn't have to be copied out. But looking up an arbitrary element is slow.

A deque classically stands for "double-ended queue", but in STL means a combination of a vector and a list. Indexing of arbitrary elements is supported, as are list operations like efficiently popping the front item off a list.

To illustrate these notions, we will go through three examples. The first one is the same as given in the last newsletter issue, and shows how a vector might be used to store a list of 25 numbers and then shuffle them into random order:

```
    #include <vector>
    #include <algorithm>
    #include <iostream>

    using namespace std;

    int main()
    {
        vector<int> v;
```

```
        for (int i = 0; i < 25; i++)
            v.push_back(i);

        random_shuffle(v.begin(), v.end());

        for (int j = 0; j < 25; j++)
            cout << v[j] << " ";
        cout << endl;

        return 0;
    }
```

With lists, we can't use [] to index the list, nor is random_shuffle() supported for lists. So we make do with:

```
    #include <list>
    #include <algorithm>
    #include <iostream>

    using namespace std;

    int main()
    {
        list<int> v;

        for (int i = 0; i < 25; i++)
            v.push_back(i);

        //random_shuffle(v.begin(), v.end());

        for (int j = 0; j < 25; j++) {
            cout << v.front() << " ";
            v.pop_front();
        }

        cout << endl;

        return 0;
    }
```

where we add elements to the list, and then simply retrieve the element at the front of the list, print it, and pop it off the list.

Finally, we present a hybrid using deques. random_shuffle() can be used with these, because they have properties of vectors. But we can also use list operations like front() and pop_front():

```
    #include <algorithm>
    #include <iostream>
    #include <deque>

    using namespace std;
```

```
int main()
{
	deque<int> v;

	for (int i = 0; i < 25; i++)
		v.push_back(i);

	random_shuffle(v.begin(), v.end());

	for (int j = 0; j < 25; j++) {
		cout << v.front() << " ";
		v.pop_front();
	}

	cout << endl;

	return 0;
}
```

Which of vectors, lists, and deques you should use depend on the application, of course. There are several additional container types that we'll be looking at in future issues, including stacks and queues. It's also possible to define your own container types.
The performance of operations on these structures is defined in the standard, and can be relied upon when designing for portability.

INTRODUCTION TO STL PART 3 - SETS

In the last issue we talked about several STL container types, namely vectors, lists, and deques. STL also has set and multiset, where set is a collection of unique values, and multiset is a set with possible non-unique values, that is, keys (elements) of the set may appear more than one time. Sets are maintained in sorted order at all times.
To illustrate the use of sets, consider the following example:

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
	typedef set<int, less<int> > SetInt;
	//typedef multiset<int, less<int> > SetInt;

	SetInt s;

	for (int i = 0; i < 10; i++) {
		s.insert(i);
		s.insert(i * 2);
```

```
        }

        SetInt::iterator iter = s.begin();

        while (iter != s.end()) {
            cout << *iter << " ";
            iter++;
        }

        cout << endl;

        return 0;
    }
```

This example is for set, but the usage for multiset is almost identical. The first item to consider is the line:

```
typedef set<int, less<int> > SetInt;
```

This establishes a type "SetInt", which is a set operating on ints, and which uses the template "less<int>" defined in <function> to order the keys of the set. In other words, set takes two type arguments, the first of which is the underlying type of the set, and the second a template class that defines how ordering is to be done in the set.

Next, we use insert() to insert keys in the set. Note that some duplicate keys will be inserted, for example "4".

Then we establish an iterator pointing at the beginning of the set, and iterate over the elements, outputting each in turn. The code for multiset is identical save for the typedef declaration.

The output for set is:

```
0 1 2 3 4 5 6 7 8 9 10 12 14 16 18
```

and for multiset:

```
0 0 1 2 2 3 4 4 5 6 6 7 8 8 9 10 12 14 16 18
```

STL also provides bitsets, which are packed arrays of binary values. These are not the same as "vector<bool>", which is a vector of Booleans.

INTRODUCTION TO STL PART 4 - MAPS

In the previous issue we talked a bit about STL sets. In this issue we'll discuss another data structure, maps. A map is something like an associative array or hash table, in that each element consists of a key and an associated value. A map must have unique keys, whereas with a multimap keys may be duplicated.

To see how maps work, let's look at a simple application that counts word frequency. Words are input one per line and the total count of each is output.

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
```

```
        typedef map<string, long, less<string> > MAP;
        typedef MAP::value_type VAL;

        MAP counter;

        char buf[256];

        while (cin >> buf)
            counter[buf]++;

        MAP::iterator it = counter.begin();

        while (it != counter.end()) {
            cout << (*it).first << " " << (*it).second << endl;
            it++;
        }

        return 0;
    }
```

This is a short but somewhat tricky example. We first set up a typedef for:

```
    map<string, long, less<string> >
```

which is a map template with three template arguments. The first is the type of the key, in this example a string. The second is the value associated with the key, in this case a long integer used as a counter. Finally, because the keys of the map are maintained in sorted order, we provide a template comparison function (see issue #016 for another example of this).

Another typedef we establish but do not use in this simple example is the VAL type, which is a template of type "pair<string,long>". pair is used internally within STL, and in this case is used to represent a map element key/value pair. So VAL represents an element in the map.

We then read lines of input and insert each word into the map. The statement:

```
    counter[buf]++;
```

does several things. First of all, buf is a char*, not a string, and must be converted via a constructor. What we've said is equivalent to:

```
    counter[string(buf)]++;
```

operator[] is overloaded for maps, and in this case the key is used to look up the element, and return a long&, that is, a reference to the underlying value. This value is then incremented (it started at zero).

Finally, we iterate over the map entries, using an iterator. Note that:

```
    (*it).first
```

cannot be replaced by:

```
    it->first
```

because "*" is overloaded. When * is applied to "it", it returns a pair<string,key> object, that is, the underlying type of elements in the map. We then reference "first" and "second", fields in pair, to retrieve keys and values for output.

For input:

```
    a
    b
    c
```

```
a
b
```

output is:

```
a 2
b 2
c 1
```

There are some complex ideas here, but map is a very powerful feature worth mastering.

INTRODUCTION TO STL PART 5 - BIT SETS

We've been looking at various types of data structures found in the Standard Template Library. Another one of these is bit sets, offering space-efficient support for sets of bits. Let's look at an example:

```
#include <iostream>
#include <bitset>

using namespace std;

int main()
{
	bitset<16> b1("1011011110001011");
	bitset<16> b2;

	b2 = ~b1;

	for (int i = b2.size() - 1; i >= 0; i--)
		cout << b2[i];
	cout << endl;

	return 0;
}
```

A declaration like:

```
bitset<16> b1("1011011110001011");
```

declares a 16-long set of bits, and initializes the value of the set to the specified bits.
We then operate on the bit set, in this example performing a bitwise NOT operation, that is, toggling all the bits. The result of this operation is stored in b2.
Finally, we iterate over b2 and display all the bits. b2.size() returns the number of bits in the set, and the [] operator is overloaded to provide access to individual bits.
There are other operations possible on bit sets, for example the flip() function to toggle an individual bit.

INTRODUCTION TO STL PART 6 - STACKS

We're nearly done discussing the basic data structures underlying the Standard Template Library. One more worth mentioning is stacks. In STL a stack is based on a vector, deque, or list. An example of stack usage is:

```
#include <iostream>
```

```
#include <stack>
#include <list>

using namespace std;

int main()
{
	stack<int, list<int> > stk;

	for (int i = 1; i <= 10; i++)
		stk.push(i);

	while (!stk.empty()) {
		cout << stk.top() << endl;
		stk.pop();
	}

	return 0;
}
```

We declare the stack, specifying the underlying type (int), and the sort of list used to represent the stack (list<int>).
We then use push() to push items on the stack, top() to retrieve the value of the top item on the stack, and pop() to pop items off the stack. empty() is used to determine whether the stack is empty or not.
We will move on to other aspects of STL in future issues. One data structure not discussed is queues and priority_queues. A queue is something like a stack, except that it's first-in-first-out instead of last-in-first-out.

INTRODUCTION TO STL PART 7 - ITERATORS

In previous issues we've covered various STL container types such as lists and sets. With this issue we'll start discussing iterators. Iterators in STL are mechanisms for accessing data elements in containers and for cycling through lists of elements.
Let's start by looking at an example:

```
#include <algorithm>
#include <iostream>

using namespace std;

const int N = 100;

void main()
{
	int arr[N];

	arr[50] = 37;
```

```
        int* ip = find(arr, arr + N, 37);
        if (ip == arr + N)
            cout << "item not found in array\n";
        else
            cout << "found at position " << ip - arr << "\n";
    }
```

In this example, we have a 100-long array of ints, and we want to search for the location in the array where a particular value (37) is stored. To do this, we call find() and specify the starting point ("arr") and ending point ("arr + N") in the array, along with the value to search for (37). An index is returned to the value in the array, or to one past the end of the array if the value is not found. In this example, "arr", "arr + N", and "ip" are iterators.

This approach works fine, but requires some knowledge of pointer arithmetic in C++. Another approach looks like this:

```
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

const int N = 100;

void main()
{
    vector<int> iv(N);

    iv[50] = 37;

    vector<int>::iterator iter = find(iv.begin(), iv.end(), 37);
    if (iter == iv.end())
        cout << "not found\n";
    else
        cout << "found at " << iter - iv.begin() << "\n";
}
```

This code achieves the same end, but is at a higher level. Instead of an actual array of ints, we have a vector of ints, and vector is a higher-level construct than a primitive C/C++ array. For example, a vector has within in it knowledge of how long it is, so that we can say "iv.end()" to refer to the end of the array, without reference to N.

In future issues we will be looking at several additional examples of iterator usage.

INTRODUCTION TO STL PART 8 - ADVANCE() AND DISTANCE()

In the last issue we started discussing iterators. They are used in the Standard Template Library to provide access to the contents of data structures, and to cycle across multiple data elements.

We presented two examples of iterator usage, the first involving pointers, the second a higher-level construct. Both of these examples require some grasp of pointer arithmetic, a daunting

subject. There's another way to write the example we presented before, using a couple of STL iterator functions:

```
#include <algorithm>
#include <iterator>
#include <vector>
#include <iostream>

using namespace std;

const int N = 100;

void main()
{
	vector<int> iv(N);

	iv[50] = 37;
	iv[52] = 47;

	vector<int>::iterator iter = find(iv.begin(), iv.end(), 37);
	if (iter == iv.end()) {
		cout << "not found\n";
	}
	else {
		int d = 0;
		distance(iv.begin(), iter, d);
		//cout << "found at " << iter - iv.begin() << "\n";
		cout << "found at " << d << "\n";
	}

	advance(iter, 2);
	cout << "value = " << *iter << "\n";
}
```

The function distance() computes the distance between two iterator values. In this example, we know that we're starting at "iv.begin()", the beginning of the integer vector. And we've found a match at "iter", and so we can use distance() to compute the distance between these, and display this result. Note that more recently distance() has been changed to work more like a regular function, with the beginning and ending arguments supplied and the difference returned as the result of the function:

```
d = distance(iv.begin(), iter);
```

A similar issue comes up with advancing an iterator. For example, it's possible to use "++" for this, but cumbersome when you wish to advance the iterator a large value. Instead of ++, advance() can be used to advance the iterator a specified number of positions. In the example above, we move the iterator forward 2 positions, and then display the value stored in the vector at that location.

These functions provide an alternative way of manipulating iterators, that does not depend so much on pointer arithmetic.

INTRODUCTION TO STL PART 9 - SORTING

We've spent the last couple of issues discussing STL iterators, which are used to access data structures. We will now start discussing some of the actual STL algorithms that can be applied to data structures. One of these is sorting.
Consider a simple example of a String class, and a vector of Strings:

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <assert>
#include <string>

class String {
	char* str;
public:
	String()
	{
		str = 0;
	}
	String(char* s)
	{
		str = strdup(s);
		assert(str);
	}
	int operator<(const String& s) const
	{
		return strcmp(str, s.str) < 0;
	}
	operator char*()
	{
		return str;
	}
};

using namespace std;

char* list[] = {"epsilon", "omega", "theta", "rho",
	"alpha", "beta", "phi", "gamma", "delta"};

const int N = sizeof(list) / sizeof(char*);

int main()
{
	int i, j;

	vector<String> v;
```

```
        for (i = 0; i < N; i++)
            v.push_back(String(list[i]));

        random_shuffle(v.begin(), v.end());

        for (j = 0; j < N; j++)
            cout << v[j] << " ";
        cout << endl;

        sort(v.begin(), v.end());

        for (j = 0; j < N; j++)
            cout << v[j] << " ";
        cout << endl;

        return 0;
    }
```

This String class provides a thin layer over char* pointers. It is provided for illustrative purposes rather than as a model of how to write a good String class.
We first build a vector of String objects by iterating over the char* list, calling a String constructor for each entry in turn. Then we shuffle the list, display it, and then sort it by calling sort() with a couple of iterator parameters v.begin() and v.end(). Output looks like:

```
phi delta beta theta omega alpha rho gamma epsilon
alpha beta delta epsilon gamma omega phi rho theta
```

There are a couple of things to note about this example. If we commented out the operator< function, the example would still compile, and the < comparison would be done by converting both Strings to char* using the conversion function we supplied. Comparing actual pointers, that is, comparing addresses, is probably not going to work, except by chance in a case where the list of char* is already in sorted order.
Also, sort() is not stable, which means that the order of duplicate items is not preserved. "stable_sort" can be used if this property is desired.
In the next few issues, we'll be looking at some of the other algorithms found in STL.

INTRODUCTION TO STL PART 10 - COPYING

In the last issue we started discussing the standard STL algorithms that are available, giving an example of sorting. Another of these is copying, which we can illustrate with a simple example:

```
#include <algorithm>
#include <iostream>

using namespace std;

int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0, 0};
int b[13];

int main()
```

```
{
	int i = 0;

	copy(a, a + 13, b);
	for (i = 0; i < 13; i++)
		cout << b[i] << " ";
	cout << endl;

	copy_backward(a, a + 10, a + 13);
	for (i = 0; i < 13; i++)
		cout << a[i] << " ";
	cout << endl;

	copy(b, b + 10, b + 3);
	for (i = 0; i < 13; i++)
		cout << b[i] << " ";
	cout << endl;

	return 0;
}
```

In the first case, we want to copy the contents of "a" to "b". We specify a couple of iterators "a" and "a + 10" to describe the region to be copied, and another iterator "b" that describes the beginning of the destination region. In the second example, we do a similar thing, except we copy backwards starting with the ending iterator. copy_backward() is important when source and destination overlap. In the third example, we copy a vector to itself, sort of a "rolling" copy. The results of running this program are:

```
1 2 3 4 5 6 7 8 9 10 0 0 0
1 2 3 1 2 3 4 5 6 7 8 9 10
1 2 3 1 2 3 1 2 3 1 2 3 1
```

As with previous examples, we could replace primitive arrays with vector<int> types, and use begin() and end() as higher-level iterator mechanisms.

Copying is a low-level, efficient operation. It does no checking while copying, so, for example, if the destination array is too small, then copying will run off the end of the array.

INTRODUCTION TO STL PART 11 - REPLACING

In the last issue we illustrated how one can do copying using STL algorithms. In this issue we'll talk about replacing a bit, that is, how to substitute elements in a data structure by use of iterators and algorithms that perform the actual replacement.

The first example uses replace() and replace_copy():

```
#include <algorithm>
#include <iostream>

using namespace std;

int vec1[10] = {1, 2, 10, 5, 9, 10, 3, 2, 7, 10};
int vec2[10] = {1, 2, 10, 5, 9, 10, 3, 2, 7, 10};
```

```
int vec3[10];

int main()
{
	int i = 0;

	replace(vec1, vec1 + 10, 10, 20);

	for (i = 0; i < 10; i++)
		cout << vec1[i] << " ";
	cout << endl;

	replace_copy(vec2, vec2 + 10, vec3, 10, 20);

	for (i = 0; i < 10; i++)
		cout << vec2[i] << " ";
	cout << endl;
	for (i = 0; i < 10; i++)
		cout << vec3[i] << " ";
	cout << endl;

	return 0;
}
```

In both cases we replace all values of "10" in the vectors with the value "20". replace_copy() is like replace(), except that the replacing is not done in place, but instead is sent to a specified location described by an iterator (in this case, "vec3").
The output of this program is:

```
1 2 20 5 9 20 3 2 7 20
1 2 10 5 9 10 3 2 7 10
1 2 20 5 9 20 3 2 7 20
```

A more general form of replacement uses replace_if(), along with a specified predicate template instance:

```
#include <algorithm>
#include <iostream>

using namespace std;

int vec1[10] = {1, 2, 10, 5, 9, 10, 3, 2, 7, 10};

template <class T> class is_odd : public unary_function<T, bool>
{
public:
	bool operator()(const T& x)
	{
		return (x % 2) != 0;
	}
};
```

```
int main()
{
	int i = 0;

	replace_if(vec1, vec1 + 10, is_odd<int>(), 59);

	for (i = 0; i < 10; i++)
		cout << vec1[i] << " ";
	cout << endl;

	return 0;
}
```

In this example, is_odd<T> is a class template that is used to determine whether a value of type T is even or odd. The constructor call, is_odd<int>(), creates an object instance of the template where T is "int". replace_if() calls operator() of the template object to evaluate whether a given value should be replaced.
This program replaces odd values with the value 59. Output is:

```
59 2 10 59 59 10 59 2 59 10
```

There is also replace_copy_if(), which combines replace_copy() and replace_if() functions.

INTRODUCTION TO STL PART 12 - FILLING

In a similar vein to some of our previous STL examples, here is an illustration of how to fill a data structure with a specified value:

```
#include <algorithm>
#include <iostream>

using namespace std;

int vec1[10];
int vec2[10];

int main()
{
	fill(vec1, vec1 + 10, -1);
	for (int i = 0; i < 10; i++)
		cout << vec1[i] << " ";
	cout << endl;

	fill_n(vec2, 5, -1);
	for (int j = 0; j < 10; j++)
		cout << vec2[j] << " ";
	cout << endl;

	return 0;
}
```

fill() fills according to the specified iterator range, while fill_n() fills a specified number of locations based on a starting iterator and a count. The results of running this program are:

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 0 0 0 0 0
```

INTRODUCTION TO STL PART 13 - ACCUMULATING

Another simple algorithm that STL makes available is accumulation, for example summing a set of numeric values. An example of this would be:

```
#include <iostream>
#include <numeric>

using namespace std;

int vec[] = {1, 2, 3, 4, 5};

int main()
{
	int sum = accumulate(vec, vec + 5, 0);

	cout << sum << endl;

	int prod = accumulate(vec, vec + 5, 1, times<int>());

	cout << prod << endl;

	return 0;
}
```

In this example, we specify iterators for a vector of integers, along with an initial value (0) for doing the summation.
By default, the "+" operator is applied to the values in turn. Other operators can be used, for example "*" in the second example. In this case the starting value is 1 rather than 0.

INTRODUCTION TO STL PART 14 - OPERATING ON SETS

We've been looking at various algorithms that can be used on STL containers. Another group of these are some algorithms for operating on ordered sets of data, illustrated by a simple example:

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int set1[] = {1, 2, 3};
int set2[] = {2, 3, 4};
vector<int> set3(10);
```

```
int main()
{

    vector<int>::iterator first = set3.begin();

    vector<int>::iterator last =
       set_union(set1, set1 + 3, set2, set2 + 3, first);

    while (first != last) {
        cout << *first << " ";
        first++;
    }
    cout << endl;

    return 0;
}
```

In the example we set up two ordered sets of numbers, and then take their union. We specify two pairs of iterators to delimit the input sets, along with an output iterator.
Algorithms are provided for taking union, intersection, difference, and for determining whether one set of elements is a subset of another set.

Exception Handling

INTRODUCTION TO EXCEPTION HANDLING PART 1 - A SIMPLE EXAMPLE

In this and subsequent issues we will be discussing some aspects of C++ exception handling. To start this discussion, let's consider a simple example. Suppose that you are writing a program to manipulate calendar dates, and want to check whether a given year is in the 20th century (ignoring the issue of whether the 21st century starts in 2000 or 2001!).
Using exceptions, one way to do this might be:

```
#include <iostream.h>

class DateException {
    char* err;
public:
    DateException(char* s) {err = s;}
    void print() const {cerr << err << endl;}
};

// a function that operates on dates
void g(int date)
```

```
{
    if (date < 1900)
        throw DateException("date < 1900");
    if (date > 1999)
        throw DateException("date > 1999");
    // process date ...
}

// some code that uses dates
void f()
{
    g(1879);
}

int main()
{
    try {
        f();
    }
    catch (const DateException& de) {
        de.print();
        return 1;
    }

    return 0;
}
```

The basic idea here is that we have a try block:

```
try {
    f();
}
```

Within this block, we execute some code, in this case a function call f(). Then we have a list of one or more handlers:

```
catch (DateException de) {
    de.print();
    return 1;
}
```

If an abnormal condition arises in the code, we can throw an exception:

```
if (date < 1900)
    throw DateException("date < 1900");
```

and have it caught by one of the handlers at an outer level, that is, execution will continue at the point of the handler, with the execution stack unwound.

An exception may be a class object type such as DateException, or a fundamental C++ type like an integer. Obviously, a class object type can store and convey more information about the nature of the exception, as illustrated in this example. Saying:

```
throw -37;
```

will indeed throw an exception, which may be caught somewhere, but this idiom is not particularly useful.

What if the handler we declare is changed slightly, as in:

```
catch (DateException* de) {
    de->print();
    return 1;
}
```

In this case, because an object of type DateException is thrown, rather than a DateException* (pointer), no corresponding handler will be found in the program. In that case, the runtime system that handles exception processing will call a special library function terminate(), and the program will abort. One way to avoid this problem is to say:

```
main()
{
    try {
        body_of_program();
    }
    catch (...) {
        // all exceptions go through here
        return 1;
    }

    return 0;
}
```

where "..." will catch any exception type.
We will explore various details of exception handling in future issues, but one general comment is in order. C++ exceptions are not the same as low-level hardware interrupts, nor are they the same as UNIX signals such as SIGTERM. And there's no linkage between exceptions such as divide by zero (which may be a low-level machine exception) and C++ exceptions.

INTRODUCTION TO EXCEPTION HANDLING PART 2 - THROWING AN EXCEPTION

In the last issue we introduced C++ exception handling. In this issue we'll go more into detail about throwing exceptions.
Throwing an exception transfers control to an exception handler. For example:

```
void f()
{
    throw 37;
}

void g()
{
    try {               // try block
        f();
    }
    catch (int i) {     // handler or catch clause
    }
}
```

In this example the exception with value 37 is thrown, and control passes to the handler. A throw transfers control to the nearest handler with the appropriate type. "Nearest" means in the sense of stack frames and try blocks that have been dynamically entered.
Typically an exception that is thrown is of class type rather than a simple constant like "37". Throwing a class object instance allows for more sophisticated usage such as conveying additional information about the nature of an exception.
A class object instance that is thrown is treated similarly to a function argument or operand in a return statement. A temporary copy of the instance may be made at the throw point, just as temporaries are sometimes used with function argument passing. A copy constructor if any is used to initialize the temporary, with the class's destructor used to destruct the temporary. The temporary persists as long as there is a handler being executed for the given exception. As in other parts of the C++ language, some compilers may be able in some cases to eliminate the temporary.
An example:

```
#include <iostream.h>

class Exc {
    char* s;
public:
    Exc(char* e) {s = e; cerr << "ctor called\n";}
    Exc(const Exc& e) {s = e.s; cerr << "copy ctor called\n";}
    ~Exc() {cerr << "dtor called\n";}
    char* geterr() const {return s;}
};

void check_date(int date)
{
    if (date < 1900)
        throw Exc("date < 1900");

    // other processing
}

int main()
{
    try {
        check_date(1879);
    }
    catch (const Exc& e) {
        cerr << "exception was: " << e.geterr() << "\n";
    }

    return 0;
}
```

If you run this program, you can trace through the various stages of throwing the exception, including the actual throw, making a temporary copy of the class instance, and the invocation of the destructor on the temporary.

It's also possible to have "throw" with no argument, as in:

```
catch (const Exc& e) {
    cerr << "exception was: " << e.geterr() << "\n";
    throw;
}
```

What does this mean? Such usage rethrows the exception, using the already-established temporary. The exception thrown is the most recently caught one not yet finished. A caught exception is one where the parameter of the catch clause has been initialized, and for which the catch clause has not yet been exited.
So in the example above, "throw;" would rethrow the exception represented by "e". Because there is no outer catch clause to catch the rethrown exception, a special library function terminate() is called. If an exception is rethrown, and there is no exception currently being handled, terminate() is called as well.
In the next issue we'll talk more about how exceptions are handled in a catch clause.

INTRODUCTION TO EXCEPTION HANDLING PART 3 - STACK UNWINDING

In the last issue we talked about throwing exceptions. Before discussing how exceptions are handled, we need to talk about an intermediate step, stack unwinding.
The exception handling mechanism is dynamic in that a record is kept of the flow of program execution, for example via stack frames and program counter mapping tables. When an exception is thrown, control transfers to the nearest suitable handler. "nearest" in this sense means the nearest dynamically surrounding try block containing a handler that matches the type of the thrown exception. We will talk more about exception handlers in a future issue.
Transfer of control from the point at which an exception is thrown to the exception handler implies jumping out of one program context into another. What about cleanup of the old program context? For example, what about local class objects that have been allocated? Are their destructors called?
The answer is "yes". All stack-allocated ("automatic") objects allocated since the try block was entered will have their destructors invoked. Let's look at an example:

```
#include <iostream.h>

class A {
    int x;
public:
    A(int i) {x = i; cerr << "ctor " << x << endl;}
    ~A() {cerr << "dtor " << x << endl;}
};

void f()
{
    A a1(1);

    throw "this is a test";

    A a2(2);
}
```

```
int main()
{
	try {
		A a3(3);

		f();

		A a4(4);
	}
	catch (const char* s) {
		cerr << "exception: " << s << endl;
	}

	return 0;
}
```

Output of this program is:

```
ctor 3
ctor 1
dtor 1
dtor 3
exception: this is a test
```

In this example, we enter the try block in main(), allocate a3, then call f(). f() allocates a1, then throws an exception, which will transfer control to the catch clause in main().
In this example, the a1 and a3 objects have their destructors called. a2 and a4 do not, because they were never allocated.
It's possible to have class objects containing other class objects, or arrays of class objects, with partial construction taking place followed by an exception being thrown. In this case, only the constructed subobjects will be destructed.

INTRODUCTION TO EXCEPTION HANDING PART 4 - HANDLING AN EXCEPTION

In previous issues we discussed throwing of exceptions and stack unwinding. Let's now look at actual handling of an exception that has been thrown. An exception is handled via an exception handler. For example:

```
catch (T x) {
	// stuff
}
```

handles exceptions of type T. More precisely, a handler of the form:

```
catch (T x) {
	// stuff
}
```

or:

```
catch (const T x) {
	// stuff
}
```

or:

```
catch (T& x) {
    // stuff
}
```

or:

```
catch (const T& x) {
    // stuff
}
```

will catch a thrown exception of type E, given that:

- T and E are the same type, or

- T is an unambiguous public base class of E, or

- T is a pointer type and E is a pointer type that can be converted to T by a standard pointer conversion

As an example of these rules, in the following case the thrown exception will be caught:

```
#include <iostream.h>

class A {};

class B : public A {};

void f()
{
    throw B();
}

int main()
{
    try {
        f();
    }
    catch (const A& x) {
        cout << "exception caught" << endl;
    }

    return 0;
}
```

because A is a public base class of B. Handlers are tried in order of appearance. If, for example, you place a handler for a derived class after a handler for a corresponding base class, it will never be invoked. If we had a handler for B after A, in the example above, it would not be called.

A handler like:

```
catch (...) {
    // stuff
```

```
}
```

appearing as the last handler in a series, will match any exception type.
If no handler is found, the search for a matching handler continues in a dynamically surrounding try block. If no handler is found at all, a special library function terminate() is called, typically ending the program.
An exception is considered caught by a handler when the parameters to the handler have been initialized, and considered finished when the handler exits.
In the next issue we'll talk a bit about exception specifications, that are used to specify what exception types a function may throw.

INTRODUCTION TO EXCEPTION HANDLING PART 5 - TERMINATE() AND UNEXPECTED()

Suppose that you have a bit of exception handling usage, like this:

```
void f()
{
	throw -37;
}

int main()
{
	try {
		f();
	}
	catch (char* s) {
	}

	return 0;
}
```

What will happen? An exception of type "int" is thrown, but there is no handler for it. In this case, a special function terminate() is called. terminate() is called whenever the exception handling mechanism cannot find a handler for a thrown exception. terminate() is also called in a couple of odd cases, for example when an exception occurs in the middle of throwing another exception.
terminate() is a library function which by default aborts the program. You can override terminate if you want:

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

typedef void (*PFV)(void);

PFV set_terminate(PFV);

void t()
{
	cerr << "terminate() called" << endl;
	exit(1);
```

```
}

void f()
{
	throw -37;
}

int main()
{
	set_terminate(t);

	try {
		f();
	}
	catch (char* s) {
	}

	return 0;
}
```

Note that this area is in a state of flux as far as compiler adaptation of new features. For example, terminate() should really be "std::terminate()", and the declarations may be found in a header file "<exception>". But not all compilers have this yet, and these examples are written using an older no-longer-standard convention.

In a similar way, a call to the unexpected() function can be triggered by saying:

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

typedef void (*PFV)(void);

PFV set_unexpected(PFV);

void u()
{
	cerr << "unexpected() called" << endl;
	exit(1);
}

void f() throw(char*)
{
	throw -37;
}

int main()
{
	set_unexpected(u);

	try {
```

```
            f();
        }
        catch (int i) {
        }

        return 0;
    }
```

unexpected() is called when a function with an exception specification throws an exception of a type not listed in the exception specification for the function. In this example, f()'s exception specification is:

```
    throw(char*)
```

A function declaration without such a specification may throw any type of exception, and one with:

```
    throw()
```

is not allowed to throw exceptions at all. By default unexpected() calls terminate(), but in certain cases where the user has defined their own version of unexpected(), execution can continue.

There is also a brand-new library function:

```
    bool uncaught_exception();
```

that is true from the time after completion of the evaluation of the object to be thrown until completion of the initialization of the exception declaration in the matching handler. For example, this would be true during stack unwinding (see newsletter #017). If this function returns true, then you don't want to throw an exception, because doing so would cause terminate() to be called.

Placement New/Delete

In C++, operators new/delete mostly replace the use of malloc() and free() in C. For example:

```
    class A {
    public:
        A();
        ~A();
    };

    A* p = new A;

    ...

    delete p;
```

allocates storage for an A object and arranges for its constructor to be called, later followed by invocation of the destructor and freeing of the storage. You can use the standard new/delete functions in the library, or define your own globally and/or on a per-class basis.

There's a variation on new/delete worth mentioning. It's possible to supply additional parameters to a new call, for example:

```
A* p = new (a, b) A;
```

where a and b are arbitrary expressions; this is known as "placement new". For example, suppose that you have an object instance of a specialized class named Alloc that you want to pass to the new operator, so that new can control allocation according to the state of this object (that is, a specialized storage allocator):

```
class Alloc {/* stuff */};

Alloc allocator;

...

class A {/* stuff */};

...

A* p = new (allocator) A;
```

If you do this, then you need to define your own new function, like this:

```
void* operator new(size_t s, Alloc& a)
{
    // stuff
}
```

The first parameter is always of type "size_t" (typically unsigned int), and any additional parameters are then listed. In this example, the "a" instance of Alloc might be examined to determine what strategy to use to allocate space. A similar approach can be used for operator new[] used for arrays.

This feature has been around for a while. A relatively new feature that goes along with it is placement delete. If during object initialization as part of a placement new call, for example during constructor invocation on a class object instance, an exception is thrown, then a matching placement delete call is made, with the same arguments and values as to placement new. In the example above, a matching function would be:

```
void operator delete(void* p, Alloc& a)
{
    // stuff
}
```

With new, the first parameter is always "size_t", and with delete, always "void*". So "matching" in this instance means all other parameters match. "a" would have the value as was passed to new earlier.

Here's a simple example:

```
int flag = 0;

typedef unsigned int size_t;

void operator delete(void* p, int i)
{
    flag = 1;
```

```
}

void* operator new(size_t s, int i)
{
	return new char[s];
}

class A {
public:
	A() {throw -37;}
};

int main()
{
	try {
		A* p = new (1234) A;
	}
	catch (int i) {
	}
	if (flag == 0)
		return 1;
	else
		return 0;
}
```

Placement delete may not be in your local C++ compiler as yet. In compilers without this feature, memory will leak. Note also that you can't call overloaded operator delete directly via the operator syntax; you'd have to code it as a regular function call.

Pointers to Members and Functions

POINTERS TO MEMBERS

In ANSI C, function pointers are used like this:

```
#include <stdio.h>

void f(int i)
{
	printf("%d\n", i);
}

typedef void (*fp)(int);
```

```
void main()
{
    fp p = &f;

    (*p)(37);      /* these are equivalent */

    p(37);
}
```

and are employed in a variety of ways, for example to specify a comparison function to a library function like qsort().

In C++, pointers can be similarly used, but there are a couple of quirks to consider. We will discuss two of them in this section, and another one in the next section.

The first point to mention is that C++ has C-style functions in it, but also has other types of functions, notably member functions. For example:

```
class A {
public:
    void f(int);
};
```

In this example, A::f(int) is a member function. That is, it operates on object instances of class A, and the function itself has a "this" pointer that points at the instance in question.

Because C++ is a strongly typed language, it is desirable that a pointer to a member function be treated differently than a pointer to a C-style function, and that a pointer to a function member of class A be distinguished from a pointer to a member of class B. To do this, we can say:

```
#include <iostream.h>

class A {
public:
    void f(int i) {cout << "value is: " << i << "\n";}
};

typedef void (A::*pmfA)(int);

pmfA x = &A::f;

void main()
{
    A a;
    A* p = &a;

    (p->*x)(37);
}
```

Note the notation for actually calling the member function.

It is not possible to intermix such a type with other pointer types, so for example:

```
void f(int) {}

pmfA x = &f;
```

is invalid.

A static member function, as in:

```
class A {
public:
    static void g(int);
};

typedef void (*fp)(int);

fp p = &A::g;
```

is treated like a C-style function. A static function has no "this" pointer and does not operate on actual object instances.
Pointers to members are typically implemented just like C function pointers, but there is an issue with their implementation in cases where inheritance is used. In such a case, you have to worry about computing offsets of subobjects, and so on, when calling a member function, and for this purpose a runtime structure similar to a virtual table used for virtual functions is used.
It's also possible to have pointers to data members of a class, with the pointer representing an offset into a class instance. For example:

```
#include <iostream.h>

class A {
public:
    int x;
};

typedef int A::*piA;
piA x = &A::x;

void main()
{
    A a;
    A* p = &a;

    a.x = 37;

    cout << "value is: " << p->*x << "\n";
}
```

Note that saying "&A::x" does not take the address of an actual data member in an instance of A, but rather computes a generic offset that can be applied to any instance.

A NEW ANGLE ON FUNCTION POINTERS

The discussion on function pointers in this issue overlooks one key angle that has fairly recently been introduced into the language. This involves distinguishing between C and C++ pointers. A C-style pointer in C++, that is, one that does not point to a member function, is

used just like a function pointer in C. But according to the standard (section 7.5), such a pointer in fact has a different type.
For example, consider:

```
extern "C" typedef void (*fp1)(int);

extern "C++" typedef void (*fp2)(int);

extern "C" void f(int);
```

fp1 and fp2 are not the same type, and saying:

```
fp2 p = &f;
```

to initialize p to the f(int) declared in the 'extern "C"' will not work.
It is possible to overload functions on this basis, so that for example:

```
extern "C" void f(void (*)(int));

extern "C++" void f(void (*)(int));
```

is legal, with the appropriate f() called based on the function pointer type passed to it. The function pointer parameter types in this example are not identical; the first is a pointer to a C function, the second a pointer to a C++ one.
This feature is new and may not be implemented in your local C++ compiler.

Type Identification

A relatively new feature in C++ is type identification, where it is possible to determine the type of an object at run time. A simple example of this feature is:

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

class A {
public:
    virtual void f(int) {}
};

class B : public A {
public:
    virtual void f(int) {}
};

int main()
{
    A a;
    B b;
    A* ap1 = &a;
```

```
        A* ap2 = &b;

        if (typeid(*ap1) == typeid(A))
            printf("ap1 is A\n");
        else
            printf("ap1 is B\n");

        if (typeid(*ap2) == typeid(A))
            printf("ap2 is A\n");
        else
            printf("ap2 is B\n");

        return 0;
    }
```

which produces:

```
    ap1 is A
    ap2 is B
```

even though the nominal type of both *ap1 and *ap2 is A. In this example, *ap1 and *ap2 represent polymorphic types, that is, types that can refer to any class type in a hierarchy of derivations. If we omitted the virtual functions in A and B, this program would give different results, considering both *ap1 and *ap2 to be referencing A objects.

typeid() produces an object of type "typeinfo", described in typeinfo.h (or just "typeinfo" in newer systems). This type has operations for testing for equality, and also a member function for returning the name of a type. For example, when this code is executed:

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

    int main()
    {
        int i;
        double x[57];
        float f1 = 0.0;
        const float f2 = 0.0;

        printf("%s\n", typeid(i).name());
        printf("%s\n", typeid(x).name());

        if (typeid(f1) == typeid(f2))
            printf("equal\n");

        return 0;
    }
```

the result is:

```
    int
    double [57]
    equal
```

Note that the typeid() comparison ignores top-level "const". The form of the name returned by name() is implementation-dependent.
This feature of C++ is quite important, because it represents a partial departure from early binding, that is, fully resolving names at compile time. Sometimes it's necessary to be able to manipulate type names in a running program. A more recent language like Java(tm) has many more features of this type.

Dynamic Casts

In the last issue we discussed runtime type identification, a mechanism for obtaining the type of an object during program execution. There is another aspect of this that we need to mention, the dynamic_cast<> feature. If we have an expression:

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

then this operator converts its operand p to the type T*, if *p really is a T or a class derived from T; otherwise, the operator returns 0.
What does this mean in practice? Suppose that you have a pointer or reference to a base class, and you want to know whether you "really" have a base class pointer, or instead a pointer to some class object derived from the base class. In this case, you can say:

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

class A {
public:
    virtual void f() {cout << "A::f\n";}
};

class B : public A {
public:
    virtual void f() {cout << "B::f\n";}
};

void f(A* p)
{
```

```
        //cout << (void*)(B*)p << endl;

        B* bp = dynamic_cast<B*>(p);
        if (bp)
            bp->f();
    }

    int main()
    {
        A* ap = new A();
        B* bp = new B();

        f(ap);
        f(bp);

        return 0;
    }
```

Here we have a program that creates A and B objects, and passes pointers to them to a function f(). f() checks whether p is really a pointer to a B, and if so, calls B::f().
Note that we could use the technique shown in the last issue if all we want to do is check the type. But there are advantages to combining the check and the cast. One is that a combined operator makes it difficult to mismatch the test and the cast. Another advantage is that a static cast, for example as illustrated in the commented-out line above, doesn't always give the correct result. That is, it relies on static information and doesn't know whether a base class pointer "really" points at a derived object instance.

Explicit Template Argument Specification

One of the newer features in C++ is the ability to explicitly specify argument types for function templates. As a simple example of this, consider the following:

```
    template <class T> void f(int i) { T x = i; ... }

    void g()
    {
        f<double>(37);
    }
```

It used to be that you'd have to use all the template parameter types (like T) in the declaration of the template, but this is no longer required. In this example, T is declared via the <> specification to be of type double, and the actual function parameter is of course an int.
One possible application for the feature is the ability to specify what a template's return type should be:

```
    template <class T, class U, class V> V max(T t, U u)
```

```
{
    if (t > u)
        return V(t);
    else
        return V(u);
}

void g()
{
    int i = max<double,double,int>(12.34, 43.21);
}
```

independent of reliance on the template function arguments.

Using Standard Libraries

INTRODUCTION TO C++ LIBRARIES PART 1 - <CASSERT> AND <CERRNO>

We are going to spend some time looking at C++ libraries, starting with a couple of library utilities used for reporting errors. These are <cassert> and <cerrno>, and work the same in C++ as in C. We will be using the older <assert.h> and <errno.h>, because the newer names don't yet exist in many compilers.

assert.h defines a macro assert() used to check assertions within a program. For example:

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

int main()
{
    FILE* fp = fopen("test.txt", "r");

    assert(fp != NULL);

    return 0;
}
```

If the argument to assert() is false (zero), the program terminates by calling the library function abort(), and gives a diagnostic as to the file and line of the error. In this example, an error like:

```
Assertion failed: fp != NULL, file x2.c, line 8
Abnormal program termination
```

comes out. Note that we could shorten the test to:

```
assert(fp);
```

identical to fp != NULL.

assert() is useful for "should never happen" kinds of errors, or for quick prototyping. It's not really suitable as a primary tool for giving end-user error messages.
Another error-reporting tool is <cerrno>. This has antecedents in the UNIX operating system, where a system call would return -1 on failure, and set a global variable "errno" to a number giving a more precise indication of what failed.
An example of using this technique is:

```
#include <stdio.h>
#include <errno.h>
#include <iostream.h>
#include <string.h>

int main()
{
	errno = 0;

	FILE* fp = fopen("test.txt", "r");

	if (fp == NULL)
		cout << strerror(errno) << endl;

	return 0;
}
```

errno is reset, and then an fopen() call made, which will ultimately invoke a system call open() to open a file. If fopen() returns NULL, errno can be queried to find out what exactly went wrong. strerror() is used to retrieve the text of the various error message codes represented by errno.
In this example, the output is:

```
No such file or directory
```

This mechanism is useful in obtaining detail about errors, but you need to be careful to reset errno each time. Also, errno is not thread-safe.

INTRODUCTION TO C++ LIBRARIES PART 2 - <STRING>

C++ inherits C-style strings from C, where a sequence of characters is terminated with a null character and referenced via a char* pointer, and storage for dynamic strings must be explicitly managed. For example:

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

int main()
{
	char buf[25];
	strcpy(buf, "testing");
	printf("%s\n", buf);
}
```

This approach works pretty well and is efficient, but is quite low-level and prone to errors.

A newer facility is C++-style strings. A simple example, that reads from standard input and writes each line to standard output, after reversing the characters in the line, looks like this:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
	string instr;

	while (cin >> instr) {
		string outstr = "";
		for (int i = instr.length() - 1; i >= 0; i--)
			outstr += instr[i];
		cout << outstr << endl;
	}

	return 0;
}
```

Note in this example that >> and << are overloaded for I/O, that [] is used to index individual characters, that += is used to concatenate an individual character to a string, and that there's no need to worry about memory management.

The string class is based on a template "basic_string" that provides string operations, and is defined as:

```
typedef basic_string<char> string;
```

But you don't need to worry about this unless you really want to make sophisticated use of string facilities. Note also that string is defined in the "std" namespace, which must be included via a using declaration.

Strings have value semantics, meaning that a copy is done when one string is assigned to another. So, for example, the output of this program:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
	string s1 = "abc";
	string s2 = s1;
	s1 = "def";

	cout << s2 << endl;

	return 0;
}
```

is "abc" and not "def".

Another example, illustrating some additional features of string, is one that replaces "abc" in input lines with "ABC", and writes the result to standard output:

```
#include <iostream>
#include <string>
#include <stdio>

using namespace std;

int main()
{
	string str;

	while (cin >> str) {
		string::size_type st = str.find("abc");
		if (st != string::npos)
			str.replace(st, 3, "ABC");
		printf("%s\n", str.c_str());
	}

	return 0;
}
```

find() attempts to find a substring in the string, and returns its index if found. "string::npos" is a special value that indicates the search failed. Strings have the property:

```
length() < npos
```

If the search succeeds, we replace "abc" with "ABC". We then output the value using C-style I/O, as an illustration of a how a C++ string can be converted to a C one using c_str().
C++ strings offer a higher-level abstraction than C-style ones, and are preferred in most cases.

INTRODUCTION TO C++ LIBRARIES PART 3 - NUMERIC_LIMITS

We've started looking at some of the features of the C++ standard library. One of these is numeric_limits, a template defined in <limits>. numeric_limits is used to obtain information about the properties of integral and floating-point types on the local system.
For example, this program:

```
#include <iostream>
#include <limits>

using namespace std;

int main()
{
	cout << numeric_limits<long>::digits << endl;
	cout << numeric_limits<double>::max_exponent10 << endl;

	return 0;
}
```

prints "31" and "308" when using Borland C++ 5.0 on a PC. 31 is the number of non-sign digits in a long, and 308 the maximum base-10 exponent of a double. Properties that do not make sense for a type (such as max_exponent10 for int) are given default values.
The set of properties that is available will vary based on the underlying type. For example, floating-point types have information available on exponents, infinity, and so on.
Some of the common properties are illustrated by another example:

```
#include <iostream>
#include <limits>

using namespace std;

int main()
{
	cout << numeric_limits<short>::is_integer << endl;
	cout << numeric_limits<short>::min() << endl;
	cout << numeric_limits<short>::max() << endl;

	return 0;
}
```

which checks whether the type (short) is an integral type, and obtains the minimum (-32768) and maximum (32767) values for the type.
If you define your own custom numeric type, it's a good idea to specialize numeric_limits for that type. For example, suppose that I have a type "LongLong" that is twice the length of a long. I might say something like:

```
#include <iostream>
#include <limits>

using namespace std;

class LongLong {/* ... */};

class numeric_limits<LongLong> {
public:
	inline static LongLong min() throw() {/* ... */}
	inline static LongLong max() throw() {/* ... */}
};

int main()
{
	cout << numeric_limits<LongLong>::min() << endl;
	cout << numeric_limits<LongLong>::max() << endl;

	return 0;
}
```

and define the appropriate members suitable for this numeric type.

INTRODUCTION TO C++ LIBRARIES PART 4 - NO-THROW OPERATOR NEW()

In previous issues we've occasionally mentioned that older versions of operator new() would return 0 if they failed, similar to what the C library function malloc() does. More recently, operator new() was specified as throwing an exception if memory could not be allocated.
In the C++ standard that was just finalized, both approaches are in fact supported. The standard operator new() throws a "bad_alloc" exception. But there's also a no-exception version, defined like this in <new>:

```
class bad_alloc : public exception {...};

struct nothrow_t {};
extern const nothrow_t nothrow;

void* operator new(size_t) throw(bad_alloc);
void* operator new(size_t, const nothrow_t&) throw();

...
```

This says that two basic flavors of new() are supported (there are also other ones such as new[] for use with arrays). The first throws a bad_alloc exception if it can't allocate memory, while the second simply returns 0. The second flavor would be used like this:

```
int* ip = new (nothrow) int[10000]; // never throws an exception
if (ip == 0)
    // allocation error
```

In other words, it's a syntactic variant of placement new() as described in issue #019.
This approach allows for error-handling strategies that do not use exceptions. Whether such strategies are "good" depends a lot on the particular application in question.

INTRODUCTION TO C++ LIBRARIES PART 5 - PROGRAM INVOCATION AND TERMINATION

In previous issues we've acknowledged that there are a variety of ways in which existing C++ compilers treat issues like the declaration of main(), program termination, and so on. It is worth revisiting this topic, given the new standard for the language.
A C++ program starts by initializing objects with static storage duration, using zero/constant initialization (as in C, and known as "static initialization"), and then performing dynamic initialization (constructors and non-constant expression initialization) for objects in the order that the objects appear in a translation unit (order between translation units is undefined).
The function main() is then called. main() is a global function that must be declared as one of:

```
int main()

int main(int argc, char* argv[])        // argc >= 0, argv[argc] == 0
```

main() cannot be predefined (such as in a library), overloaded, called within a program, inlined, or made static. The linkage of main() (for example C or C++ linkage) is implementation defined.
A "return X;" statement in main() results in the destruction of automatic objects, and works as if "exit(X)" was called. If execution falls off the end of main(), it will be treated as though a "return 0;" was the last statement.

The function abort() terminates a C++ program, without executing destructors for objects with automatic or static storage duration, and without calling functions registered with atexit().
atexit() is used to register functions that are to be called when the program terminates.
The effect of exit() is to call the destructors for all static objects, in the reverse order of their construction (automatic objects are not destructed). This last-in-first-out process also incorporates functions registered with atexit(), such that a function registered with atexit() after a static object is constructed, will be called before that static object is destructed.
After this process is complete, all open C streams are flushed and closed, files created with tmpfile() are removed, and an exit status is passed back to the calling system.
Some aspects of all the above are a little tricky, but it's important to understand the complete process of invocation and termination. Sometimes there are important parts of an application (such as stream I/O) that depend on the underlying mechanics of program startup and shutdown.

INTRODUCTION TO C++ STANDARD LIBRARIES PART 6 - STANDARD EXCEPTIONS

With the standardization of C++, there is a set of standard exceptions that have been identified. These are thrown in response to various types of error conditions, both in the standard library and in user programs.
The organization of these is:

exception

bad_alloc (out of memory)
bad_exception (called for unexpected exceptions)
bad_cast (bad operand to dynamic_cast<>)
bad_typeid (bad operand to typeid())
ios_base::failure (I/O output)

logic_error
length_error (invalid length)
domain_error (domain error)
out_of_range (argument out of range)
invalid_argument (invalid argument)

runtime_error
range_error (out of range in internal computation)
overflow_error (overflow error)
underflow_error (underflow error)

arranged in a corresponding class hierarchy.
An example of using these exceptions would be the following program:

```
#include <iostream>
#include <stdexcept>

using namespace std;

void f(int x, int y)
```

```
{
	if (x < 0)
		throw invalid_argument("x < 0");
	if (y < 0)
		throw invalid_argument("y < 0");
}

int main()
{
	try {
		f(37, -59);
	}
	catch (exception e) {
		cout << e.what() << endl;
	}

	return 0;
}
```

The thrown exception is caught in main(). If it had not been, the program would terminate. You can of course create your own exception classes, derived or not from "exception". But it's worth knowing about and using standard exceptions wherever possible.

INTRODUCTION TO C++ STANDARD LIBRARIES PART 7 - PAIR

The C++ standard library contains much support for containers and algorithms, I/O, locale support, and so on. One of the library classes (actually a template) that sort of falls between the cracks is pair<T1,T2>, defined in <utility>.
Pair is used to construct objects, which contain a pair of values of two arbitrary types T1 and T2. Pair is defined as:

```
template <class T1, class T2> struct std::pair {
	T1 first;
	T2 second;
	pair(const T1& x, const T2& y) : first(x), second(y) {}
	// omits a couple of other constructors
};
```

A related function std::make_pair() is also provided to create pairs.

This is a very simple idea, but a quite useful one. For example, consider the problem of returning two values from a function, such as the minimum and maximum of a set of values:

```
#include <algorithm>
#include <utility>
#include <iostream>

using namespace std;

template <class T> pair<T,T> minmax(T* vec, int n)
{
```

```
        return pair<T,T>(*min_element(vec, vec + n),
           *max_element(vec, vec + n));
    }

    int main()
    {
        int vec[] = {1, 19, 2, 14, -5, 59, 67, -37, 100, 47};

        pair<int,int> p = minmax(vec, 10);

        cout << p.first << " " << p.second << endl;

        return 0;
    }
```

minmax() takes a T* vector argument and a vector size, and returns the minimum and maximum values of the vector, using the library functions min_element() and max_element(). The values are passed back in a pair<T,T> structure.
Pair is used in the standard library, for example to represent the (key,value) pair within the map container.

INTRODUCTION TO C++ STANDARD LIBRARIES PART 7 - COMPLEX

A class for doing complex arithmetic is often presented as an illustration of how to design a user-defined type, and various versions of such a class exist today in actual C++ implementations.
The recently-standardized C++ library includes a complex type, as a template rather than simply a class. Basing complex on a template allows for specification of the underlying scalar type, with specializations provided for float, double, and long double. So three complex class types are guaranteed to be defined:

```
    std::complex<float>

    std::complex<double>

    std::complex<long double>
```

All the usual operations on complex types are provided by this template. For example, a simple program that multiplies two complex numbers is:

```
    #include <complex>
    #include <iostream>

    using namespace std;

    typedef complex<double> ComplexDouble;

    int main()
    {
        ComplexDouble a(1.0, 2.0);
        ComplexDouble b(3.0, 5.0);
```

```
        cout << a * b << endl;

        return 0;
    }
```

which takes the product of (1.0,2.0) and (3.0,5.0), yielding (-7.0,11.0).
Complex does not do any special error checking for domain or range errors, beyond that provided by underlying operations such as sqrt().

Operators new[] and delete[]

The C++ library has long had operator new() and delete() for dynamic storage allocation. Note that with these there's a distinction made between the operators specified as keywords, as in:

```
    new A[10];
```

and the functions, for example:

```
    operator new(159);
```

The former usage not only is responsible for allocating space, via operator new(), but also for arranging for constructors to be called for the individual objects in the array slots. So normally you will not use operator new() directly.
More recently the functions operator new[]() and operator delete[]() have been added to the language. These are like operator new() and operator delete(), but are invoked when arrays are being allocated and deallocated.
To see how this works, consider an example such as:

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

    class A {
        int x;
    public:
        A() {printf("A::A %lx\n", (unsigned long)this);}
        ~A() {printf("A::~A %lx\n", (unsigned long)this);}
    };

    void* operator new[](size_t sz)
    {
        printf("allocated size = %lu\n", (unsigned long)sz);

        void* vp = operator new(sz);
        printf("allocated pointer = %lx\n", (unsigned long)vp);

        return vp;
    }
```

```
void operator delete[](void* ptr)
{
    printf("returned pointer = %lx\n", (unsigned long)ptr);
    operator delete(ptr);
}

int main()
{
    A* ap = new A[10];
    delete [] ap;

    return 0;
}
```

This example redefines operator new[]() and operator delete[](), and they are invoked when the program is executed.

When operator new[]() is called, it is passed an argument indicating how many bytes are required for the total array. In this example, approximately 40 bytes are needed for the 10 array slots (this will vary from system to system, with overhead for each chunk of space allocated).

In the example above, the actual bytes are allocated via a call to operator new(), that is, the non-array version is called to allocate the bytes. operator delete[]() works in a similar way.

Note that the C++ standard specifies that the size of the array is saved, so that when it is deleted, the system will know how many slots to iterate across to call the destructors for individual objects.

Typical output of the program is:

```
allocated size = 44
allocated pointer = 7b2514
A::A 7b2518
A::A 7b251c
A::A 7b2520
A::A 7b2524
A::A 7b2528
A::A 7b252c
A::A 7b2530
A::A 7b2534
A::A 7b2538
A::A 7b253c
A::~A 7b253c
A::~A 7b2538
A::~A 7b2534
A::~A 7b2530
A::~A 7b252c
A::~A 7b2528
A::~A 7b2524
A::~A 7b2520
A::~A 7b251c
```

A::~A 7b2518
returned pointer = 7b2514

Note that objects are constructed and then destructed in LIFO (last-in first-out) order. Also, note that we used C-style I/O instead of stream I/O to print out information. Why is this? If stream I/O is used here, the program will crash with a popular compiler, probably because at the first call to operator new[](), the I/O system is not initialized as yet (the call to new in this case is presumably to obtain a buffer to initialize the system). So you need to be very careful in overloading the global versions of new and delete.

It's also possible to define operator new[]() and operator delete[]() on a per-class basis. Considering this feature and the one described in the next section, there are six varieties each of new and delete:

regular + throws exception

regular + doesn't throw exception

array + throws exception

array + doesn't throw exception

placement + doesn't throw exception

placement + array + doesn't throw exception

C++ Character Sets

With the recent standardization of C++, it's useful to review some of the mechanisms included in the language for dealing with character sets. This might seem like a very simple issue, but there are some complexities to contend with.

The first idea to consider is the notion of a "basic source character set" in C++. This is defined to be:

all ASCII printing characters 041 - 0177, save for @ $ ` DEL

space

horizontal tab

vertical tab

form feed

newline

or 96 characters in all. These are the characters used to compose a C++ source program.
Some national character sets, such as the European ISO-646 one, use some of these character positions for other letters. The ASCII characters so affected are:

[ ] { } | \

To get around this problem, C++ defines trigraph sequences that can be used to represent these characters:

[	??(
]	??)
{	??<
}	??>
|	??!
\	??/
#	??=
^	??'
~	??-

Trigraph sequences are mapped to the corresponding basic source character early in the compilation process.
C++ also has the notion of "alternative tokens", that can be used to replace tokens with others. The list of tokens and their alternatives is this:

{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:
&&	and
|	bitor
||	or
^	xor

~ compl

& bitand

&= and_eq

|= or_eq

^= xor_eq

! not

!= not_eq

Another idea is the "basic execution character set". This includes all of the basic source character set, plus control characters for alert, backspace, carriage return, and null. The "execution character set" is the basic execution character set plus additional implementation-defined characters. The idea is that a source character set is used to define a C++ program itself, while an execution character set is used when a C++ application is executing.
Given this notion, it's possible to manipulate additional characters in a running program, for example characters from Cyrillic or Greek. Character constants can be expressed using any of:

\137 octal

\xabcd hexadecimal

\u12345678 universal character name (ISO/IEC 10646)

\u1234 -> \u00001234

This notation uses the source character set to define execution set characters. Universal character names can be used in identifiers (if letters) and in character literals:

'\u1234'

L'\u2345'

The above features may not yet exist in your local C++ compiler. They are important to consider when developing internationalized applications.

Allocators

In previous issues, we've looked at some of the standard containers (such as vector) found in the C++ Standard Library. One of the interesting issues that comes up is how such containers manage memory. It turns out that containers use what is called a standard allocator, defined in <memory>.
To see a bit of how this works, we will devise a custom allocator:

```
#include <vector>
#include <cstddef>
#include <iostream>

using namespace std;

template <class T> class alloc {
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;

    void construct(pointer p, const_reference val)
    {
        new (p) T(val);
    }

    void destroy(pointer p)
    {
        p->~T();
    }

    pointer allocate(size_type sz, void* vp = 0)
    {
        pointer p = static_cast<pointer>(operator new(sz *
          sizeof(value_type)));

        cout << "allocate " << (unsigned long)p << endl;

        return p;
    }

    void deallocate(pointer p, size_type)
    {
        cout << "deallocate " << (unsigned long)p << endl;

        operator delete(p);
    }
};
```

This allocator overrides the default allocator for vector, and is specified by saying:

```
vector<int, alloc<int> > v;
```

The allocator template establishes a series of standard internal types such as "pointer". The real work gets done in allocate() and deallocate(), which are used to allocate N objects of type

T. The standard allocator uses operator new() and operator delete() to actually allocate/deallocate memory.
Normally you don't need to worry too much about this area, but sometimes for reasons of speed and space you may wish to construct your own allocator for use with standard containers. For example, allocators can be written that are efficient for very small objects, or that use shared memory, or that use memory from pre-allocated pools of objects.

C++ as a Better C

- Function Prototypes
- References
- Operator New/Delete
- Declaration Statements
- Function Overloading
- Operator Overloading
- Inline Functions
- Type Names
- External Linkage
- General Initializers
- Jumping Past Initialization
- Function Parameter Names
- Character Types and Arrays
- Function-style Casts
- Bit Field Types
- Anonymous Unions
- Empty Classes
- Hiding Names

Function Prototypes

People often ask about how to get started with C++ or move a project or development team to the language. There are many answers to this question. One of the simplest and best is to begin using C++ as a "better C". This term doesn't have a precise meaning but can be illustrated via a series of examples. We will cover some of these examples in forthcoming issues of the newsletter.

One simple but important area of difference between C and C++ deals with the area of function definition and invocation. In older versions of C ("Classic C"), functions would be defined in this way:

```
f(s)
char* s;
{
    return 0;
}
```

The return type of this function is implicitly "int", and the function has no prototype. In ANSI C and in C++, a similar definition would be:

```
int f(char* s)
{
    return 0;
}
```

Why does this matter? Well, suppose that you call the function with this invocation:

```
f(s)
char* s;
{
    return 0;
}

g()
{
    f(23);
}
```

In Classic C, this would be a serious programming error, because a value of integer type (23) is being passed to a function expecting a character pointer. However, the error would not be flagged by the compiler, and the result would be a runtime failure such as a crash. By contrast, in ANSI C and in C++ the compiler would flag such usage.

Very occasionally, you want to cheat, and actually pass a value like
23 as a character pointer. To do this, you can say:

```
f((char*)23);
```

Such usage is typically only seen in very low level systems programming.
Using function prototypes in C++ is a big step forward from Classic C; this approach will eliminate a large class of errors in which the wrong number or types of arguments are passed to a function.

References

In the last newsletter we discussed using function prototypes in C++ to eliminate a common type of error encountered in C, that of calling a function with the wrong number or types of arguments. Another C++ feature that can be used to reduce programming errors is known as references.

A reference is another name for an object. For example, in this code:

```
int i;

int& ir = i;
```

ir is another name for i. To see how references are useful, and also how they're implemented, consider writing a function that has two return values to pass back. In ANSI C, we might say:

```
void f(int a, int b, int* sum, int* prod)
{
    *sum = a + b;
    *prod = a * b;
}

void g()
{
    int s;
    int p;

    f(37, 47, &s, &p);
}
```

In C++, we would say:

```
void f(int a, int b, int& sum, int& prod)
{
    sum = a + b;
    prod = a * b;
}

void g()
{
    int s;
```

```
        int p;

        f(37, 47, s, p);
    }
```

One way of viewing references is to consider that they have some similarities to C pointers, but with one level of pointer removed. Pointers are a frequent source of errors in C.
A reference must be initialized, and its value (the pointed at object) cannot be changed after initialization. The value of the reference cannot change, but the value of the referenced object can, unless the reference is declared as const. So, for example:

```
    int i = 0;
    int& ir = i;
    ir = -19;                   // i gets the value -19
```

is acceptable, while:

```
    const int& irc = 47;
    irc = -37;                  // error
```

is not. A constant reference that points at a value like 47 can be implemented using a temporary.
References are useful in argument passing and return. Another use is illustrated below in the section on writing robust code.

Operator New/Delete

In the first newsletter we talked about using C++ as a better C. This term doesn't have a precise meaning, but one way of looking at it is to focus on the features C++ adds to C, exclusive of the most obvious one, namely the class concept used in object-oriented programming.
One of these features is operator new and operator delete. These are intended to replace malloc() and free() in the C standard library. To give an example of how these are similar and how they differ, suppose that we want to allocate a 100-long vector of integers for some purpose. In C, we would say:

```
    int* ip;

    ip = (int*)malloc(sizeof(int) * 100);

    ...

    free((void*)ip);
```

With new/delete in C++, we would have:

```
    int* ip;

    ip = new int[100];
```

```
...

delete ip;
```

The most obvious difference is that the C++ approach takes care of the low-level details necessary to determine how many bytes to allocate. With the C++ new operator, you simply describe the type of the desired storage, in this example "int[100]".
The C and C++ approaches have several similarities:

- neither malloc() nor new initialize the space to zeros

- both malloc() and new return a pointer that is suitably aligned for a given machine architecture

- both free() and delete do nothing with a NULL pointer

malloc() returns NULL if the space cannot be obtained. Many versions of new in existing C++ compilers do likewise. However, the draft ANSI C++ standard says that a failure to obtain storage should result in an exception being thrown or should result in the currently installed new handler being invoked. In these newsletters we are assuming that NULL is returned.
The idea of a new handler can be illustrated as follows:

```
extern "C" int printf(const char*, ...);
extern "C" void exit(int);

typedef void (*new_handler)(void);

new_handler set_new_handler(new_handler);

void f()
{
    printf("new handler invoked due to new failure\n");
    exit(1);
}

main()
{
    float* p;

    set_new_handler(f);

    for (;;)
        p = new float[5000];    // something that will
                                // fail eventually
    return 0;
}
```

A new handler is a way of establishing a hook from the C++ standard library to a user program. set_new_handler() is a library function that records a pointer to another function that is to be called in the event of a new failure.
It is possible to define your own new and delete functions. For example:

```
void* operator new(size_t s)
```

```
{
    // allocate and align storage of size s

    // handle failure via new_handler or exception

    // return pointer to storage
}

void operator delete(void* p)
{
    // handle case where p is NULL

    // handle deallocation of p block in some way
}
```

size_t is a typedef, typically defined to mean "unsigned int". It's found in a header file that may vary between compiler implementations.

(clarification of above)
In the previous issue of the newsletter, there was an example:

```
int* ip;

ip = new int[100];

delete ip;
```

This code will work with many compilers, but it should instead read:

```
int* ip;

ip = new int[100];

delete [] ip;
```

This is an area of C++ that has changed several times in recent years. There are a number of issues to note. The first is that new and delete in C++ have more than one function. The new operator allocates storage, just like malloc() in C, but it is also responsible for calling the constructor for any class object that is being allocated. For example, if we have a String class, saying:

```
String* p = new String("xxx");
```

will allocate space for a String object, and then call the constructor to initialize the String object to the value "xxx". In a similar way, the delete operator arranges for the destructor to be called for an object, and then the space is deallocated in a manner similar to the C function free().
If we have an array of class objects, as in:

```
String* p = new String[100];
```

then a constructor must be called for each array slot, since each is a class object. Typically this processing is handled by a C++ internal library function that iterates over the array.
In a similar way, deallocation of an array of class objects can be done by saying:

```
delete [] p;
```

It used to be that you had to say:

```
delete [100] p;
```

but this feature is obsolete. The size of the array is recovered by the library function that implements the delete operator for arrays. The pointer/size pair can be stored in an auxiliary data structure or the size can be stored in the allocated block before the first actual byte of data.

What makes this a bit tricky is that all of this work of calling constructors and destructors doesn't matter for fundamental data types like int:

```
int* ip;

ip = new int[100];

delete ip;
```

This code will work in many cases, because there are no destructors to call, and deleting a block of storage works pretty much the same whether it's treated as an array of ints or a single large chunk of bytes.

But more recently, the ANSI standardization committee has decided to break out the new and delete operators for arrays as separate functions, so that a program can control the allocation of arrays separately from other types. For example, you can say:

```
void* operator new(unsigned int) {/* ... */ return 0;}

void* operator new[](unsigned int) {/* ... */ return 0;}

void f()
{
    int* ip;

    ip = new int;          // calls operator new()

    ip = new int[100];     // calls operator new[]()
}
```

and the appropriate functions will be called in each case. This is kind of like defining your own versions of the malloc() and free() library functions in C.

Declaration Statements

In C, when you write a function, all the declarations of local variables must appear at the top of the function or at the beginning of a block:

```
void f()
{
    int x;
    /* ... */
    while (x) {
        int y;
```

```
            /* ... */
        }
    }
```

Each such variable has a lifetime that corresponds to the lifetime of the block it's declared in. So in this example, x is accessible throughout the whole function, and y is accessible inside the while loop.

In C++, declarations of this type are not required to appear only at the top of the function or block. They can appear wherever C++ statements are allowed:

```
    class A {
    public:
        A(double);
    };
    void f()
    {
        int x;
        /* ... */
        while (x) {
            /* ... */
        }
        int y;
        y = x + 5;
        /* ... */
        A aobj(12.34);
    }
```

and so on. Such a construction is called a "declaration statement". The lifetime of a variable declared in this way is from the point of declaration to the end of the block.

A special case is used with for statements:

```
    for (int i = 1; i <= 10; i++)
        ...

    /* i no longer available */
```

In this example the scope of i is the for statement. The rule about the scope of such variables has changed fairly recently as part of the ANSI standardization process, so your compiler may have different behavior.

Why are declaration statements useful? One benefit is that introducing variables with shorter lifetimes tends to reduce errors. You've probably encountered very large functions in C or C++ where a single variable declared at the top of the function is used and reused over and over for different purposes. With the C++ feature described here, you can introduce variables only when they're needed.

Function Overloading

Suppose that you are writing some software to manipulate calendar dates, and you wish to allow a user of the software to specify dates in one of two forms:

8, 4, 1964 (as a triple of numbers)

August 4, 1964 (as a string)

In C, if there is a function to convert a raw date into an internal form (for example, the number of days since January 1, 1800), it might look like:

```
long str_to_date(int m, int d, int y) { ... }

long str_to_date(char* d) { ... }
```

with one function for each of the two types of dates. Unfortunately, this usage is illegal in C, because two different functions cannot have the same name "str_to_date".

In C++ this usage is legal and goes by the term "function overloading". That is, two or more functions may have the same name, so long as the parameter types are sufficiently different enough to distinguish which function is intended. A function may not be overloaded on the basis of its return type.

The question of what makes two function parameter lists sufficiently different is an interesting one. For example, this usage is not valid:

```
void f(int) {}

void f(const int) {}
```

whereas saying:

```
void f(int) {}

void f(long) {}
```

is fine.

A common place where function overloading is seen is in constructors for a class. For example, we might have:

```
class Date {
    ...
public:
    Date(int m, int d, int y);
    Date(char*);
};
```

to represent a calendar date. Two constructors, representing the two ways of creating a date object (from a triple of numbers and from a string) are specified.

What can go wrong with function overloading? Consider an example of a String class:

```
class String {
    ...
public:
    String();
    String(char*);
```

```
        String(char);
    };
```

Here we have three constructors, the first to create a null String and the second to create a String from a char*. The third constructor creates a String from an individual character, so that for example 'x' turns into a String "x".

What happens if you declare a String object like this:

```
    String s(37);
```

Clearly, the first String constructor won't be called, because it takes no arguments. And 37 isn't a valid char*, so the second constructor won't be used. That leaves String(char), but 37 is an int and not a char. The third constructor will indeed be called, after 37 is demoted from an int to a char.

In this case, the user "got away" with doing things this way, though it's not clear what was intended. Usage like:

```
    String s(12345);
```

is even more problematic, because 12345 cannot be converted to a char in any meaningful way.

The process of determining which function should be called is known as "argument matching", and it's one of the most difficult aspects of C++ to understand. Function overloading is powerful, but it's smart to use it in a way that makes clear which function will be called when.

Operator Overloading

Suppose that you are using an enumeration and you wish to output its value:

```
    enum E {e = 37};

    cout << e;
```

37 will indeed be output, by virtue of the enumerator value being promoted to an int and then output using the operator<<(int) function found in iostream.h.

But what if you're interested in actually seeing the enumerator values in symbolic form? One approach to this would be as follows:

```
    #include <iostream.h>

    enum E {e1 = 27, e2 = 37, e3 = 47};

    ostream& operator<<(ostream& os, E e)
    {
        char* s;
        switch (e) {
            case e1:
                s = "e1";
                break;
```

```
            case e2:
                s = "e2";
                break;
            case e3:
                s = "e3";
                break;
            default:
                s = "badvalue";
                break;
        }
        return os << s;
    }

    main()
    {
        enum E x;

        x = e3;

        cout << x << "\n";
        cout << e1 << "\n";
        cout << e2 << "\n";
        cout << e3 << "\n";
        cout << E(0) << "\n";

        return 0;
    }
```

In the last output statement, we created an invalid enumerator value and then output it.
Operator overloading in C++ is very powerful but can be abused. It's quite possible to create a system of operators such that it is difficult to know what is going on with a particular piece of code.
Some uses of overloaded operators, such as [] for array indexing with subscript checking, -> for smart pointers, or + - * / for doing arithmetic on complex numbers, can make sense, while other uses may not.

Inline Functions

Suppose that you wish to write a function in C to compute the maximum of two numbers. One way would be to say:

```
    int max(int a, int b)
    {
        return (a > b ? a : b);
```

```
}
```

But calling a frequently-used function can be a bit slow, and so you instead use a macro:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The extra parentheses are required to handle cases like:

```
max(a = b, c = d)
```

This approach can work pretty well. But it is error-prone due to the extra parentheses and also because of side effects like:

```
max(a++, b++)
```

An alternative in C++ is to use inline functions:

```
inline int max(int a, int b)
{
    return (a > b ? a : b);
}
```

Such a function is written just like a regular C or C++ function. But it IS a function and not simply a macro; macros don't really obey the rules of C++ and therefore can introduce problems. Note also that one could use C++ templates to write this function, with the argument types generalized to any numerical type.

If an inline function is a member function of a C++ class, there are a couple of ways to write it:

```
class A {
public:
    void f() { /* stuff */ }        // "inline" not needed
};
```

or:

```
class A {
public:
    inline void f();
};

inline void A::f()
{
    /* stuff */
}
```

The second style is often a bit clearer.

The "inline" keyword is merely a hint to the compiler or development environment. Not every function can be inlined. Some typical reasons why inlining is sometimes not done include:

- the function calls itself, that is, is recursive

- the function contains loops such as for(;;) or while()

- the function size is too large

Most of the advantage of inline functions comes from avoiding the overhead of calling an actual function. Such overhead includes saving registers, setting up stack frames, and so on. But with large functions the overhead becomes less important.

Inline functions present a problem for debuggers and profilers, because the function is expanded at the point of call and loses its identity. A compiler will typically have some option available to disable inlining.

Inlining tends to blow up the size of code, because the function is expanded at each point of call. The one exception to this rule would be a very small inline function, such as one used to access a private data member:

```
class A {
    int x;
public:
    int getx() {return x;}
};
```

which is likely to be both faster and smaller than its non-inline counterpart.
A simple rule of thumb when doing development is not to use inline functions initially. After development is mostly complete, you can profile the program to see where the bottlenecks are and then change functions to inlines as appropriate.
Here's a complete program that uses inline functions as part of an implementation of bit maps. Bit maps are useful in storing true/false values efficiently. Note that in a couple of places we could use the new bool fundamental type in place of ints. Also note that this implementation assumes that chars are 8 bits in width; there's no fundamental reason they have to be (in Java(tm) the Unicode character set is used and chars are 16 bits).
This example runs about 50% faster with inlines enabled.

```
#include <assert.h>
#include <stdlib.h>
#include <string.h>

//#define inline

class Bitmap {
    typedef unsigned long UL;       // type of specified bit num
    UL len;                     // number of bits
    unsigned char* p;               // pointer to the bits
    UL size();                  // figure out bitmap size
public:
    Bitmap(UL);                 // constructor
    ~Bitmap();                  // destructor
    void set(UL);               // set a bit
    void clear(UL);             // clear a bit
    int test(UL);               // test a bit
    void clearall();            // clear all bits
};

// figure out bitmap size
inline Bitmap::UL Bitmap::size()
{
    return (len - 1) / 8 + 1;
}

// constructor
inline Bitmap::Bitmap(UL n)
{
```

```
        assert(n > 0);
        len = n;
        p = new unsigned char[size()];
        assert(p);
        clearall();
}

// destructor
inline Bitmap::~Bitmap()
{
        delete [] p;
}

// set a bit
inline void Bitmap::set(UL bn)
{
        assert(bn < len);
        p[bn / 8] |= (1 << (bn % 8));
}

// clear a bit
inline void Bitmap::clear(UL bn)
{
        assert(bn < len);
        p[bn / 8] &= ~(1 << (bn % 8));
}

// test a bit, return non-zero if set
inline int Bitmap::test(UL bn)
{
        assert(bn < len);
        return p[bn / 8] & (1 << (bn % 8));
}

// clear all bits
inline void Bitmap::clearall()
{
        memset(p, 0, size());
}

#ifdef DRIVER
main()
{
        const unsigned long N = 123456L;
        int i;
        long j;
        int k;
```

```
        int r;

        for (i = 1; i <= 10; i++) {
            Bitmap bm(N);

            // set all bits then test

            for (j = 0; j < N; j++)
                bm.set(j);
            for (j = 0; j < N; j++)
                assert(bm.test(j));

            // clear all bits then test

            for (j = 0; j < N; j++)
                bm.clear(j);
            for (j = 0; j < N; j++)
                assert(!bm.test(j));

            // run clearall() then test

            bm.clearall();
            for (j = 0; j < N; j++)
                assert(!bm.test(j));

            // set and clear random bits

            k = 1000;
            while (k-- > 0) {
                r = rand() & 0xffff;
                bm.set(r);
                assert(bm.test(r));
                bm.clear(r);
                assert(!bm.test(r));
            }
        }

        return 0;
    }
    #endif
```

Type Names

In C, a common style of usage is to say:

```
struct A {
    int x;
};
typedef struct A A;
```

after which A can be used as a type name to declare objects:

```
void f()
{
    A a;
}
```

In C++, classes, structs, unions, and enum names are automatically type names, so you can say:

```
struct A {
    int x;
};

void f()
{
    A a;
}
```

or:

```
enum E {ee};

void f()
{
    E e;
}
```

By using the typedef trick you can follow a style of programming in C somewhat like that used in C++.

But there is a quirk or two when using C++. Consider usage like:

```
struct A {
    int x;
};

int A;
```

```
void f()
{
    A a;
}
```

This is illegal because the int declaration A hides the struct declaration. The struct A can still be used, however, by specifying it via an "elaborated type specifier":

```
struct A
```

The same applies to other type names:

```
class A a;

union U u;

enum E e;
```

Taking advantage of this feature, that is, giving a class type and a variable or function the same name, isn't very good usage. It's supported for compatibility reasons with old C code; C puts structure tags (names) into a separate namespace, but C++ does not. Terms like "struct compatibility hack" and "1.5 namespace rule" are sometimes used to describe this feature.

External Linkage

One of the common issues that always comes up with programming languages is how to mix code written in one language with code written in another.

For example, suppose that you're writing C++ code and wish to call C functions. A common case of this would be to access C functions that manipulate C-style strings, for example strcmp() or strlen(). So as a first try, we might say:

```
extern size_t strlen(const char*);
```

and then use the function. This will work, at least at compile time, but will probably give a link error about an unresolved symbol.

The reason for the link error is that a typical C++ compiler will modify the name of a function or object ("mangle" it), for example to include information about the types of the arguments. As an example, a common scheme for mangling the function name strlen(const char*) would result in:

```
strlen__FPCc
```

There are two purposes for this mangling. One is to support function overloading. For example, the following two functions cannot both be called "f" in the object file symbol table:

```
int f(int);

int f(double);
```

But suppose that overloading was not an issue, and in one compilation unit we have:

```
extern void f(double);
```

and we use this function, and its name in the object file is just "f". And suppose that in another compilation unit the definition is found, as:

```
void f(char*) {}
```

This will silently do the wrong thing -- a double will be passed to a function requiring a char*. Mangling the names of functions eliminates this problem, because a linker error will instead be triggered. This technique goes by the name "type safe linkage".

So to be able to call C functions, we need to disable name mangling. The way of doing this is to say:

```
extern "C" size_t strlen(const char*);
```

or:

```
extern "C" {
    size_t strlen(const char*);
    int strcmp(const char*, const char*);
}
```

This usage is commonly seen in header files that are used both by C and C++ programs. The extern "C" declarations are conditional based on whether C++ is being compiled instead of C. Because name mangling is disabled with a declaration of this type, usage like:

```
extern "C" {
    int f(int);
    int f(double);
}
```

is illegal (because both functions would have the name "f").

Note that extern "C" declarations do not specify the details of what must be done to allow C++ and C code to be mixed. Name mangling is commonly part of the problem to be solved, but only part.

There are other issues with mixing languages that are beyond the scope of this presentation. The whole area of calling conventions, such as the order of argument passing, is a tricky one. For example, if every C++ compiler used the same mangling scheme for names, this would not necessarily result in object code that could be mixed and matched.

General Initializers

In C, usage like:

```
int f() {return 37;}

int i = 47;

int j;
```

for global variables is legal. Typically, in an object file and an executable program these types of declarations might be lumped into sections with names like "text", "data", and "bss", meaning "program code", "data with an initializer", and "data with no initializer".

When a program is loaded by the operating system for execution, a common scheme will have the text and data stored within the binary file on disk that represents the program, and the bss

section simply stored as an entry in a symbol table and created and zeroed dynamically when the program is loaded.
There are variations on this scheme, such as shared libraries, that are not our concern here. Rather, we want to discuss the workings of an extension that C++ makes to this scheme, namely general initializers for globals. For example, I can say:

```
int f() {return 37;}

int i = 47;

int j = f() + i;
```

In some simple cases a clever compiler can compute the value that should go into j, but in general such values are not computable at compile time. Note also that sequences like:

```
class A {
public:
    A();
    ~A();
};

A a;
```

are legal, with the global "a" object constructed before the program "really" starts, and destructed "after" the program terminates.
Since values cannot be computed at compile time, they must be computed at run time. How is this done? One way is to generate a dummy function per object file:

```
int f() {return 37;}

int i = 47;

int j; // = f() + i;

static void __startup()
{
    j = f() + i;
}
```

and a similar function for shutdown as would be needed for calling destructors. Using a small tool that will modify binaries, and an auxiliary data structure generated by the compiler, it's possible to link all these __startup() function instances together in a linked list, that can be traversed when the program starts.
Typically this is done by immediately generating a call from within main() to a C++ library function _main() that iterates over all the __startup() functions. On program exit, similar magic takes place, typically tied to exit() function processing. This approach is used in some compilers but is not required; the standard mandates "what" rather than "how".
Some aspects of this processing have precedent in C. For example, when a program starts, standard I/O streams stdin, stdout, and stderr are established for doing I/O.
Within a given translation unit (source file), objects are initialized in the order of occurrence, and destructed in reverse order (last in first out). No ordering is imposed between files.
Some ambitious standards proposals have been made with regard to initialization ordering, but none have caught on. The draft standard says simply that all static objects in a translation unit

(objects that persist for the life of the program) are zeroed, then constant initializers are applied (as in C), then dynamic general initializers are applied "before the first use of a function or object defined in that translation unit".
Calling the function abort() defined in the standard library will terminate the program without destructors for global static objects being called. Note that some libraries, for example stream I/O, rely on destruction of global class objects as a hook for flushing I/O buffers. You should not rely on any particular order of initialization of global objects, and using a startup() function called from main(), just as in C, still can make sense as a program structuring mechanism for initializing global objects.

Jumping Past Initialization

As we've seen in several examples in previous newsletters, C++ does much more with initializing objects than C does. For example, class objects have constructors, and global objects can have general initializers that cannot be evaluated at compile time.
Another difference between C and C++ is the restriction C++ places on transferring control past an initialization. For example, the following is valid C but invalid C++:

```
#include <stdio.h>

int main()
{
	goto xxx;

	{
		int x = 0;
xxx:
		printf("%d\n", x);
	}

	return 0;
}
```

With one compiler, compiling and executing this program as C code results in a value of 512 being printed, that is, garbage is output. Thus the restriction makes sense.
The use of goto statements is best avoided except in carefully structured situations such as jumping to the end of a block. Jumping over initializations can also occur with switch/case statements.

Function Parameter Names

Suppose that you have a C++ function, and for some reason you don't actually use all the function parameters:

```
int sum(int a, int b, int c)
{
    return a + b;   // c not used
}
```

Many compilers will give a warning in this case to the effect "warning: parameter c not used". This is perfectly legal code but the warning can be tedious to deal with.

C++ has a feature that allows you to simply omit the parameter name:

```
int sum(int a, int b, int)
{
    return a + b;
}
```

and avoid the warning. This feature is especially handy when stubbing out code. A similar feature exists in catch handlers used in exception handling.

Character Types and Arrays

There are a couple of differences in the way that ANSI C and C++ treat character constants and arrays of characters. One of these has to do with the type of a character constant. For example:

```
#include <stdio.h>

int main()
{
    printf("%d\n", sizeof('x'));

    return 0;
}
```

If this program is compiled as ANSI C, then the value printed will be sizeof(int), typically 2 on PCs and 4 on workstations. If the program is treated as C++, then the printed value will be sizeof(char), defined by the draft ANSI/ISO standard to be 1. So the type of a char constant in C is int, whereas the type in C++ is char. Note that it's possible to have sizeof(char) == sizeof(int) for a given machine architecture, though not very likely.
Another difference is illustrated by this example:

```
#include <stdio.h>

char buf[5] = "abcde";
```

```
int main()
{
	printf("%s\n", buf);

	return 0;
}
```

This is legal C, but invalid C++. The string literal requires a trailing \0 terminator, and there is not enough room in the character array for it. This is valid C, but you access the resulting array at your own risk. Without the terminating null character, a function like printf() may not work correctly, and the program may not even terminate.

Function-style Casts

In C and C++ (and Java(tm)), you can cast one object type to another by usage like:

```
double d = 12.34;

int i = (int)d;
```

Casting in this way gets around type system checking. It may introduce problems such as loss of precision, but is useful in some cases.
In C++ it's possible to employ a different style of casting using a functional notation:

```
double d = 12.34;

int i = int(d);
```

This example achieves the same end as the previous one.
The type of a cast using this notation is limited. For example, saying:

```
unsigned long*** p = unsigned long***(0);
```

is invalid, and would need to be replaced by:

```
typedef unsigned long*** T;

T p = T(0);
```

or by the old style:

```
unsigned long*** p = (unsigned long***)0;
```

Casting using functional notation is closely tied in with constructor calls. For example:

```
class A {
public:
	A();
	A(int);
};

void f()
{
```

```
        A a;
        a = A(37);
    }
```

causes an A object local to f() to be created via the default constructor. Then this object is assigned the result of constructing an A object with 37 as its argument. In this example there is both a cast (of sorts) and a constructor call. If we want to split hairs a perhaps more appropriate technical name for this style of casting is "explicit type conversion".
It is also possible have usage like:

```
    void f()
    {
        int i;

        i = int();
    }
```

If this example used a class type with a default constructor, then the constructor would be called both for the declaration and the assignment. But for a fundamental type, a call like int() results in a zero value of the given type. In other words, i gets the value 0.
The reason for this feature is to support generality when templates are used. There may be a template such as:

```
    template <class T> class A {
        void f()
        {
            T t = T();
        }
    };
```

and it's desirable that the template work with any sort of type argument.
Note that there are also casts of the form "static_cast<T>" and so on, which we will discuss in a future issue.

Bit Field Types

Here's a small difference between C and C++. In ANSI C, bit fields must be of type "int", "signed int", or "unsigned int". In C++, they may be of any integral type, for example:

```
enum E {e1, e2, e3};

class A {
public:
    int x : 5;
    unsigned char y : 8;
    E z : 5;
};
```

This extension was added in order to allow bit field values to be passed to functions expecting a particular type, for example:

```
void f(E e)
{
}

void g()
{
    A a;
    a.z = e3;
    f(a.z);
}
```

Note that even with this relaxation of C rules, bit fields can be problematic to use. There are no pointers or references to bit fields in C++, and the layout and size of fields is tricky and not necessarily portable.

Anonymous Unions

Here's a simple one. In C++ this usage is legal:

```
struct A {
    union {
        int x;
        double y;
        char* z;
    };
};
```

whereas in C you'd have to say:

```
struct A {
    union {
        int x;
        double y;
        char* z;
```

```
        } u;
    };
```

giving the union a name. With the C++ approach, you can treat the union members as though they were members of the enclosing struct.
Of course, the members still belong to the union, meaning that they share memory space and only one is active at a given time.

Empty Classes

Here's a simple one. In C, an empty struct like:

```
    struct A {};
```

is invalid, whereas in C++ usage like:

```
    struct A {};
```

or:

```
    class B {};
```

is perfectly legal. This type of construct is useful when developing a skeleton or placeholder for a class.
An empty class has size greater than zero. Two class objects of empty classes will have distinct addresses, as in:

```
    class A {};

    void f()
    {
        A* p1 = new A;
        A* p2 = new A;

        // p1 != p2 at this point ...
    }
```

There are still one or two C++ compilers that generate C code as their "assembly" language. To handle an empty class, they will generate a dummy member, so for example:

```
    class A {};
```

becomes:

```
    struct A {
        char __dummy;
    };
```

in the C output.

Hiding Names

Consider this small example:

```
#include <stdio.h>

int xxx[10];

int main()
{
	struct xxx {
		int a;
	};

	printf("%d\n", sizeof(xxx));

	return 0;
}
```

When compiled as C code, it will typically print a value like 20 or 40, whereas when treated as C++, the output value will likely be 2 or 4.

Why is this? In C++, the introduction of the local struct declaration hides the global "xxx", and the program is simply taking the size of a struct which has a single integer member in it. In C, "sizeof(xxx)" refers to the global array, and a tag like "xxx" doesn't automatically refer to a struct.

If we said "sizeof(struct xxx)" then we would be able to refer to the local struct declaration.

C++ Performance

Handling a Common strcmp() Case

In this section of the newsletter we will present some practical performance tips for improving code speed and reducing memory usage. Some of these tips will be useful only for C++ code and some will be more general and applicable to C or other languages.
As a first example, consider an application using C-style strings and functions such as strcmp(). A recent experience with this sort of application involved a function that does word stemming, that is, takes words such as "motoring" and reduces them to their root stem, in this case "motor".
In profiling this function, it was observed that much of the overall time was being spent in the strcmp() function. For the C++ compiler in question (Borland 3.1), this function is written in assembly language and is quite fast, and attempts to speed it up by unrolling the equivalent code locally at the point of function call will typically result in slowing things down.

But it's still the case that calling a function, even one implemented in assembly language, has some overhead, which comes from saving registers, manipulating stack frames, actual transfer of control, and so on. So it might be worth trying to exploit a common case -- the case where you can determine the relationship of the strings by looking only at the first character.
So we might use an inline function in C++ to encapsulate this logic:

```
inline int local_strcmp(const char* s, const char* t)
{
    return (*s != *t ? *s - *t : strcmp(s, t));
}
```

If the first characters of each string do not match, there's no need to go further by calling strcmp(); we already know the answer.
Another way to implement the same idea is via a C macro:

```
#define local_strcmp(s, t) ((s)[0] != (t)[0] ? (s)[0] - (t)[0] : \
    strcmp((s), (t)))
```

This approach has a couple of disadvantages, however. Macros are hard to get right because of the need to parenthesize arguments so as to avoid subtly wrong semantics. Writing local_strcmp() as a real function is more natural.
And macros are less likely to be understood by development tools such as browsers or debuggers. Inline functions are also a source of problems for such tools, but they at least are part of the C++ language proper, and many C++ compilers have a way of disabling inlining to help address this problem.
How much speedup is this approach good for? In the word stemming program, for input of about 65000 words, the times in seconds were:

```
strcmp()              9.7

inline local_strcmp()   7.5

#define local_strcmp()  7.5
```

or a savings of about 23%. Obviously, this figure will vary with the compiler and the application.
This particular speedup is achieved by exploiting a common case -- the case where the first letters of two strings are different. For applications involving English words, this is often a good assumption. For some other types of strings, it may not be.

Handling Lots of Small Strings With a C++ Class

In performance tips this issue, we will present a complete C++ class along with its implementation code, which is appended to the end of the discussion.
This C++ example addresses a common problem in applications that use a lot of small strings. The problem has to do with the overhead associated with allocating the string via malloc() (in C) or operator new() (C++). Typically, such overhead is 8-16 bytes per string. And allocating then deallocating many small blocks will tend to fragment memory.

The Copy class deals with the problem by internally allocating large blocks and then shaving off small chunks for individual strings. It keeps track of all the large blocks allocated and deallocates them when a given Copy object is no longer needed. To use this system, you would allocate a Copy object for each major subsystem in your application that uses small strings. For example, at one point in your application, you might need to read in a dictionary from disk and use it for a while. You would allocate a Copy object and then use it to allocate the strings for each word, then flush the strings all at once.
In the application that this class was devised for, implementing string copying in this way saved 50K out of a total available memory pool of 500K. This is with Borland C++, which rounds the number of requested bytes for a string to the next multiple of 16, or an average wastage of 8 bytes. Since the Copy class uses 1024-byte chunks, on average 512 bytes will be wasted for a given set of strings, so the breakeven point would be 512 / 8 = 64 or more strings.
There are many variations on this theme. For example, if you are certain that the strings will never be freed, then you can simply grab a large amount of memory and shave chunks off of it, without worrying about keeping track of the allocated memory. Or if you have many objects of one class, such as tree nodes, you can overload operator new() for that class to do a similar type of thing.
Note that this particular storage allocator is not general. The allocated storage is aligned on 1-byte boundaries. This means that trying to allocate other than char* objects may result in performance degradation or a memory fault (such as "bus error" on UNIX systems). And the performance gains of course decline somewhat with large strings, while the wastage increases from stranding parts of the 1024-byte allocated chunks.
This same approach could be used in C or assembly language, but C++ makes it easier and encourages this particular style of programming.
An example of usage is included. A dictionary of 20065 words with total length 168K is read in. Without use of the Copy class it requires 354K, an 111% overhead. With the Copy class it takes 194K, an overhead of 15%. This is a difference of 160K, or 8 bytes per word. The results will of course vary depending on a particular operating system and runtime library. And the Copy version runs about 20% faster than the conventional version on a 486 PC.
The driver program that is included will work only with Borland C++, so you will need to write some other code to emulate the logic.

```
#include <string.h>
#include <assert.h>

const int COPY_BUF = 1024; // size of buffer to get const int COPY_VEC = 64; // starting size of vector
class Copy {
    int ln;                     // number of buffers in use
    int maxln;                  // max size of vector
    char** vec;                 // storage vector
    int freelen;                // length free in current
    char* freestr;              // current free string
public:
    Copy();                     // constructor
```

```
        ~Copy();                        // destructor
        char* copy(char*);              // copy a string
};
// constructor
Copy::Copy()

{
        ln = 0;
        maxln = 0;
        vec = 0;
        freelen = 0;
        freestr = 0;
}
// destructor
Copy::~Copy()

{       int i;

        // delete buffers

        for (i = 0; i < ln; i++)
                delete vec[i];

        // delete vector itself

        if (vec)
                delete vec;
}
// copy a string char* Copy::copy(char* s) {
        int i;
        char** newvec;
        int len;
        char* p;

        assert(s && *s);

        len = strlen(s) + 1;

        // first time or current buffer exhausted?

        if (len > freelen) {
                if (!vec || ln == maxln) {

                        // reallocate vector

                        maxln = (maxln ? maxln * 2 : COPY_VEC);
                        newvec = new char*[maxln];
```

```
                assert(newvec);
                for (i = 0; i < ln; i++)
                        newvec[i] = vec[i];
                if (vec)
                        delete vec;
                vec = newvec;
        }

        // allocate new buffer

        vec[ln] = new char[COPY_BUF];
        assert(vec[ln]);
        freelen = COPY_BUF;
        freestr = vec[ln];
        ln++;
    }

    // allocate and copy string

    freelen -= len;
    p = freestr;
    freestr += len;
    strcpy(p, s);
    return p;
}


#ifdef DRIVER

#include <stdio.h>
#include <alloc.h>

main() {
    long cl;
    const int MAXLINE = 256;
    char buf[MAXLINE];
    FILE* fp;
    char* s;
#ifdef USE_COPY
    Copy co;
#endif
    cl = coreleft();
    fp = fopen("c:/src/words", "r");
    assert(fp);
    while (fgets(buf, MAXLINE, fp) != NULL) {
#ifdef USE_COPY
        s = co.copy(buf);
```

```
#else
        s = new char[strlen(buf) + 1];
        assert(s);
        strcpy(s, buf);
#endif
    }
    fclose(fp);
    printf("memory used = %ld\n", cl - coreleft());

    return 0;
}
#endif
```

Hidden Constructor/Destructor Costs

Consider a short example of C++ code:

```
class A {
    int x, y, z;
public:
    A();
};

class B {
    A a;
public:
    B() {}
};

A::A() {x = 0; y = 0; z = 0;}
```

Class A has a constructor A::A(), used to initialize three of the class's data members. Class B has a constructor declared inline (defined in the body of the class declaration). The constructor is empty.

Suppose that we use a lot of B class objects in a program. Each object must be constructed, but we know that the constructor function body is empty. So will there be a performance issue? The answer is possibly "yes", because the constructor body really is NOT empty, but contains a call to A::A() to construct the A object that is part of the B class. Direct constructor calls are not used in C++, but conceptually we could think of B's constructor as containing this code:

```
B::B() {a.A::A();}      // construct "a" object in B class
```

There's nothing sneaky about this way of doing things; it falls directly out of the language definition. But in complex cases, such as ones involving multiple levels of inheritance, a seemingly empty constructor or destructor can in fact contain a large amount of processing.

Declaration Statements

Suppose that you have a function to compute factorials (1 x 2 x ... N):

```
double fact(int n)
{
    double f = 1.0;
    int i;
    for (i = 2; i <= n; i++)
        f *= (double)i;
    return f;
}
```

and you need to use this factorial function to initialize a constant in another function, after doing some preliminary checks on the function parameters to ensure that all are greater than zero. In C you can approach this a couple of ways. In the first, you would say:

```
/* return -1 on error, else 0 */
int f(int a, int b)
{
	const double f = fact(25);

	if (a <= 0 || b <= 0)
		return -1;

	/* use f in calculations */

	return 0;
}
```

This approach does an expensive computation each time, even under error conditions. A way to avoid this would be to say:

```
/* return -1 on error, else 0 */
int f(int a, int b)
{
	const double f = (a <= 0 || b <= 0 ? 0.0 : fact(25));

	if (a <= 0 || b <= 0)
		return -1;

	/* use f in calculations */

	return 0;
}
```

but the logic is a bit torturous. In C++, using declaration statements (see above), this problem can be avoided entirely, by saying:

```
/* return -1 on error, else 0 */
int f(int a, int b)
{
	if (a <= 0 || b <= 0)
		return -1;

	const double f = fact(25);

	/* use f in calculations */

	return 0;
}
```

Stream I/O Performance

Is stream I/O slower than C-style standard I/O? This question is a bit hard to answer. For a simple case like:

```
#ifdef CPPIO
#include <iostream.h>
#else
#include <stdio.h>
#endif

main()
{
        long cnt = 1000000L;

        while (cnt-- > 0)
#ifdef CPPIO
                cout << 'x';
#else
                putchar('x');
#endif
        return 0;
}
```

the C++ stream I/O approach is about 50% slower for a couple of popular C++ compilers. But putchar() is a macro (equivalent to an inline function) that has been tuned, whereas the C++ functions in iostream.h are less tuned, and in the 50% slower case not all the internal little helper functions are actually inlined. We will say more about C++ function inlining some other time, but one of the issues with it is trading space for speed, that is, doing a lot of inlining can drive up code size.
And 50% may be irrelevant unless I/O is a bottleneck in your program in the first place.

Stream I/O Output

In issue #006 we talked about stream I/O, and an example like this was shown:

```
cout << x << "\n";
```

A couple of people wrote. One said that:

```
cout << x << endl;
```

was preferable, while another said:

```
cout << x << '\n';
```

would be a better choice on performance grounds, that is, output a single character instead of a C string containing a single character.
Using one popular C++ compiler (Borland C++ 4.52), and outputting 100K lines using these three methods, the running times in seconds are:

```
"\n"        1.9

'\n'        1.3

endl        13.2
```

Outputting a single character is a little simpler than outputting a string of characters, so it's a bit faster.
Why is endl much slower? It turns out that it has different semantics. Besides adding a newline character like the other two forms do, it also flushes the output buffer. On a UNIX-like system, this means that ultimately a write() system call is done for each line, an expensive operation. Normally, output directed to a file is buffered in chunks of size 512 or 1024 or similar.
The Borland compiler has a #define called _BIG_INLINE_ in iostream.h that was enabled to do more inlining and achieve the times listed here.
Does this sort of consideration matter very much? Most of the time, no. If you're doing interactive I/O, it is best to write in the style that is plainest to you and others. If, however, you're writing millions of characters to files, then you ought to pay attention to an issue like this.
Note also that there's no guarantee that performance characteristics of stream I/O operations will be uniform across different compilers. It's probably true in most cases that outputting a single character is cheaper than outputting a C string containing a single character, but it doesn't have to be that way.

Per-class New/Delete

Some types of applications tend to use many small blocks of space for allocating nodes for particular types of data structures, small strings, and so on. In issue #002 we talked about a technique for efficiently allocating many small strings.
Another way of tackling this problem is to overload the new/delete operators on a per-class basis. That is, take over responsibility for allocating and deallocating the storage required by class objects. Here is an example of what this would look like for a class A:

```
#include <stddef.h>
#include <stdlib.h>

class A {
    int data;
    A* next;
#ifdef USE_ND
```

```
        static A* freelist;          // pointer to free list
#endif
public:
        A();
        ~A();
#ifdef USE_ND
        void* operator new(size_t);     // overloaded new()
        void operator delete(void*);    // overloaded delete()
#endif
};

#ifdef USE_ND
A* A::freelist = 0;

inline void* A::operator new(size_t sz)
{
        // get free node from freelist if any

        if (freelist) {
                A* p = freelist;
                freelist = freelist->next;
                return p;
        }

        // call malloc() otherwise

        return malloc(sz);
}

inline void A::operator delete(void* vp)
{
        A* p = (A*)vp;

        // link freed node onto freelist

        p->next = freelist;
        freelist = p;
}
#endif

A::A() {}

A::~A() {}

#ifdef DRIVER
const int N = 1000;
A* aptr[N];
```

```
int main()
{
	int i;
	int j;

	// repeatedly allocate / deallocate A objects

	for (i = 1; i <= N; i++) {
		for (j = 0; j < N; j++)
			aptr[j] = new A();
		for (j = 0; j < N; j++)
			delete aptr[j];
	}

	return 0;
}
#endif
```

We've also included a driver program. For this example, that recycles the memory for object instances, the new approach is about 4-5X faster than the standard approach.
When new() is called for an A type, the overloaded function checks the free list to see if any old recycled instances are around, and if so one of them is used instead of calling malloc(). The free list is shared across all object instances (the freelist variable is static). delete() simply returns a no-longer-needed instance to the free list.
This technique is useful only for dynamically-created objects. For static or local objects, the storage has already been allocated (on the stack, for example).
We have again sidestepped the issue of whether a failure in new() should throw an exception instead of returning an error value. This is an area in transition in the language.
There are other issues with writing your own storage allocator. For example, you have to make sure that the memory for an object is aligned correctly. A double of 8-byte length may need to be aligned, say, on a 4-byte boundary for performance reasons or to avoid addressing exceptions ("bus error - core dumped" on a UNIX system). Other issues include fragmentation and support for program threads.

Duplicate Inlines

Suppose that you have a bit of code such as:

```
inline long fact(long n)
{
	if (n < 2)
```

```
            return 1;
        else
            return n * fact(n - 1);
    }

    int main()
    {
        long x = fact(23);

        return 0;
    }
```

to compute the factorial function via a recursive algorithm. Will fact() actually be expanded as an inline? In many compilers, the answer is no. The "inline" keyword is simply a hint to the compiler, which is free to ignore it.

So what happens if the inline function is not expanded as inline? The answer varies from compiler to compiler. The traditional approach is to lay down a static copy of the function body, one copy for each translation unit where the inline function is used, and with such copies persisting throughout the linking phase and showing up in the executable image. Other approaches lay down a provisional copy per translation unit, but with a smart linker to merge the copies.

Extra copies of functions in the executable can be quite wasteful of space. How do you avoid the problem? One way is to use inlines sparingly at first, and then selectively enable inlining based on program profiling that you've done. Just because a function is small, with a high call overhead at each invocation, doesn't necessarily mean that it should be inline. For example, the function may be called only rarely, and inlining might not make any difference to the total program execution time.

Another approach diagnoses the problem after the fact. For example, here's a simple script that finds duplicate inlines on UNIX systems:

```
    #!/bin/sh

    nm $@ |
    egrep ' t ' |
    awk '{print $3}' |
    sort |
    uniq -c |
    sort -nr |
    awk '$1 > = 2{print}' |
    demangle
```

nm is a tool for dumping the symbol tables of objects or executables. A " t " indicates a static text (function) symbol. A list of such symbols is formed and those with a count of 2 or more filtered out and displayed after demangling their C++ names ("demangle" has various names on different systems).

This technique is simply illustrative and not guaranteed to work on every system.

Note also that some libraries, such as the Standard Template Library, rely heavily on inlining. STL is distributed as a set of header files containing inline templates, with the idea being that the inlines are expanded per translation unit.

Much of the time such an approach is perfectly acceptable, but it's worth at least knowing what's going on behind the scenes with inlining, and what you can do about it if performance is not acceptable.

Writing Robust Code

- Assert and Subscript Checking
- Constructors and Integrity Checking
- Stream I/O

Assert and Subscript Checking

Many of the techniques used in writing robust C code also apply in C++. For example, if you have a function that is supposed to be passed a person's name, as a C-style string, it would be wise to say:

```
#include <assert.h>

void f(char* name)
{
    assert(name && *name);

    ...
}
```

to perform basic checks on the passed-in pointer. assert() is a function (actually a macro) that checks whether its argument is true (non-zero), and aborts the program if not.
But C++ offers additional opportunities to the designer interested in producing quality code. For example, consider a common problem in C, where vector bounds are not checked during a dereference operation, and a bad location is accessed or written to.
In C++, you can partially solve this problem by defining a Vector class, with a vector dereferencing class member defined for the Vector, and the vector size stored:

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

```
class Vector {
      int len;              // number of elements
      int* ptr;             // pointer to elements
public:
      Vector(int);          // constructor
      ~Vector();            // destructor
      int& operator[](int);   // dereferencing
};
//constructor
Vector::Vector(int n)

{
      assert(n >= 1);

      len = n;              // store length
      ptr = new int[n];       // get storage to store elements
      assert(ptr);
}
//destructor
Vector::~Vector()

{
      delete ptr;

}
//dereferencing int& Vector::operator[](int i) {
      assert(i >= 1 && i <= len);

      return ptr[i - 1];      // return reference to vector slot
}
//driver program main() {
      int i;
      const int N = 10;
      Vector v(N);

      for (i = 1; i <= N; i++)       // correct usage
            v[i] = i * i;

      for (i = 1; i <= N; i++)       // correct usage
            printf("%d %d\n", i, v[i]);

      v[0] = 0;                    // will trigger assert failure

      return 0;
}
```

In this example, we create a vector of 10 elements, and the vector is indexed 1..10. If the vector is dereferenced illegally, as in:

```
v[0] = 0;
```

an assertion failure will be triggered.
One objection to this technique is that it can be slow. If every vector reference requires a function call (to Vector::operator[]), then there may be a large performance hit. However, performance concerns can be dealt with by making the dereferencing function inline.
Two other comments about the above example. We are assuming in these newsletters that if operator new() fails, it returns a NULL pointer:

```
ptr = new int[n];
assert(ptr);          // check for non-NULL pointer
```

The current draft ANSI standard says that when such a failure occurs, an exception is thrown or else a new handler is invoked. Because many C++ implementations still use the old approach of returning NULL, we will stick with it for now.
The other comment concerns the use of references. In the code:

```
v[i] = i * i;
```

the actual code is equivalent to:

```
v.operator[](i) = i * i;
```

and could actually be written this way (see a C++ reference book on operator overloading for details).
Vector::operator[] returns a reference, which can be used on the left-hand side of an assignment expression. In C the equivalent code would be more awkward:

```
#include <stdio.h>
int x[10]; // use f() to index into x[10]
int* f(int i) {
    return &x[i - 1];
}
main() {
    *f(5) = 37;

    printf("%d %d\n", *f(5), x[4]);

    return 0;
}
```

Constructors and Integrity Checking

Imagine that you want to devise a way to represent calendar dates for use in a C program. You come up with a struct:

```
struct Date {
    int month;
    int day;
    int year;
};
```

and a program using the Date struct can initialize a struct like so:

```
struct Date d;

d.month = 9;
d.day = 25;
d.year = 1956;
```

And you devise various functions, for example one to compute the number of days between two dates:

```
long days_b_dates(struct Date* d1, struct Date* d2);
```

This approach can work pretty well.
But what happens if someone says:

```
struct Date d;

d.month = 9;
d.day = 31;
d.year = 1956;
```

and then calls a function like days_b_dates()? The date in this example is invalid, because month 9 (September) has only 30 days. Once an invalid date is introduced, functions that use the date will not work properly. In C, one way to deal with this problem would be to have a function to do integrity checking on each Date pointer passed to a function like days_b_dates().
In C++, a simpler and cleaner approach is to use a constructor to ensure the validity of an object. A constructor is a function called when an object comes into scope. So I could say:

```
#include <assert.h>

class Date {
    int month;
    int day;
    int year;
    static int isleap(int);
public:
    Date(int, int, int);
};

const char days_in_month[12] = {31, 28, 31, 30, 31, 30,
   31, 31, 30, 31, 30, 31};

// return 1 if year is a leap year, else 0
int Date::isleap(int y)
{
    if (y % 4)
        return 0;
    if (y % 100)
        return 1;
    if (y % 400)
        return 0;
    return 1;
```

```
}

// constructor for Date class
Date::Date(int m, int d, int y)
{
      assert(m >= 1 && m <= 12);
      assert(d >= 1);
      assert(y >= 1800 && y <= 2099);
      assert(d <= days_in_month[m-1] ||
         (m == 2 && d == 29 && isleap(y)));

      month = m;
      day = d;
      year = y;
}

      Date d(9, 25, 1956);
```

This logic does a complete check of the date. It ensures that a Date object has internal integrity. Note that the three data members of the Date object are private to the class, meaning that a random user of a Date class object cannot change them, and instead must rely on the constructor for setting the value of a Date object.

Stream I/O

Suppose that you wish to output three values and you use some C-style output to do so:

```
printf("%d %d %d\n", a, b);
```

What is wrong here? Well, the output specification calls for three integer values to be output, but only two were specified. You can probably "get away" with this usage without your program crashing, with the printf() routine picking up a garbage value from the stack. But many cases of this usage will crash the program.

A similar case would be:

```
printf("%d %d %d\n", a, b, c, d);
```

which is even more likely to work, with the extra argument ignored. This problem is intrinsic to printf() and related functions.

Using stream I/O as illustrated above eliminates this particular problem completely:

```
cout << a << " " << b << " " << c << "\n";
```

as well as the related problem illustrated by:

```
int a;

printf("%s\n", a);
```

where the argument is of the wrong type. Stream I/O is fundamentally safer than C-style I/O; stream I/O is said to be "type safe".

Miscellaneous Topics

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Throughout this document, when the Java trademark appears alone it is a reference to the Java programming language. When the JDK trademark appears alone it is a reference to the Java Development Kit.

Standard Template Library

STL is a C++ library that has been voted by ANSI to be part of the C++ standard library.
We normally think of software libraries as being sets of functions like printf() or malloc(). But the STL library is different. It's a collection of C++ templates, that is, parameterized types. For example, there is a template for manipulating lists, and you can have lists of ints or doubles or void*** pointers or class A objects. This is what is meant by "template" or "parameterized type". A template is kind of like a C++ class except that you can specify parameters to the template to indicate what types it should operate upon.
The virtue of this approach is that one can implement an algorithm once, say for sorting a list, and then use the algorithm for any types of data that would be in a list -- numbers, strings, class objects, and so on.
Templates are a little bit like C macros, and STL is distributed as a set of header files.
Combining a template with particular argument types is a process known as "instantiation", and a template bound to particular arguments is known as a "template class". So, for example,

```
template <class T> class List { ... };
```

is a List template declaration, and

```
List<double> dlist;
```

declares a variable "dlist" of the template class List<double>, which is instantiated by supplying the type argument "double" to the List template.
You can find out more about STL via this Web site:

http://www.cs.rpi.edu/~musser/stl.html

or download an implementation of the library via FTP from:

butler.hpl.hp.com

C++ and Java(tm)

You may have heard recently of the programming language Java(tm), being pushed by Sun Microsystems as the language for Internet programming on the World Wide Web. There is a lot of hoopla about this at present. However, it's interesting to look at Java simply as a language, divorced from its Internet context. C++ was based on C, and Java is based at least in part on C++.

Giving a detailed comparison of the languages is beyond the scope of the newsletter, but if you wish to find out more, there are several places to look. Sun has a Web site:

http://java.sun.com

with useful information in it, and an anonymous FTP site as well:

java.sun.com

Another Web site with pointers to many Java resources is:

http://www.gamelan.com

I've also looked at the book "Java!" by Tim Ritchey, which appears to have a lot of useful information that gives some context to the language and its use. There are many more Java books in the works that will be appearing in the next few months.

Book Review - The Mythical Man-Month

Some of you may have heard of the book "The Mythical Man-Month" by Fred Brooks. It was first published in 1975 and was updated last year. It is published by Addison-Wesley and costs about $25, and is considered a classic with 250,000 copies in print. Brooks was the manager of the project that developed OS/360 during the early 1960s.

The subject of the book is software development and the complexities associated with it. In the earlier chapters, which have not been updated since the 1975 edition, he talks about a variety of issues. One of my favorite parts is in the first chapter, entitled "The Tar Pit". In the discussion in this chapter he distinguishes four stages in the evolution of a finished software product:

(1) a program

(2) a programming system, with interfaces and system integration

(3) a programming product, with generalization, testing, documentation, and maintenance

(4) a programming systems product

A program is something you might quickly put together in a few hours or days or weeks. But to take the additional two steps of coming up with a programming system or programming

product is a lot of additional work, on the order of 3X as Brooks describes it. Each of these steps is independent, therefore Brooks talks about a 9X ratio of cost between a program and a programming systems product.
Of course, 9X isn't a magic figure, but it captures the huge difference in cost between hacking out a few thousand lines of code over the weekend and putting out a polished product to customers.
The book has been updated with significant new material. He discusses the promise and practicality of object-oriented programming, software reuse, and so on.
Highly recommended.

Calendar Date Class

As a means of illustrating what an actual large and complete C++ class looks like, we will present a class for managing calendar dates. Commentary on this class is given below the source.
First of all, the header:

```
// Date class header file

#ifndef __DATE_H__
#define __DATE_H__

typedef unsigned short Drep;          // internal storage format

const int MIN_YEAR = 1875;
const int MAX_YEAR = 2025;
const Drep MAX_DAY = 55152;
const int DOW_MIN = 6;

class Date {
    Drep d;                          // actual date
    static int init_flag;            // init flag
    static int isleap(int);          // leap year?
    static Drep cdays[MAX_YEAR-MIN_YEAR+1]; // cumul days per yr
    static void init_date();         // initialize date
    static Drep mdy_to_d(int, int, int);    // m/d/y --> day
    static void d_to_mdy(Drep,int&,int&,int&);// day --> m/d/y
public:
    Date(Drep);                  // constructor from internal
    Date(const Date&);           // copy constructor
    Date(int, int, int);         // constructor from m/d/y
    Date(const char*);           // constructor from char*
    operator Drep();             // conversion to Drep
```

```
        void print(char* = (char*)0);   // print
        void get_mdy(int&, int&, int&); // get m/d/y
        long operator-(const Date&);    // difference of dates
        int dow();                      // day of week
        long wdays(const Date&);        // work days between dates
    };

    #endif
```

and then the source itself, along with a driver program:

```
    // Date class and driver program

    #include <string.h>
    #include <stdlib.h>
    #include <stdio.h>
    #include <ctype.h>
    #include <assert.h>
    #include "date.h"

    // days in the various months
    const char days_in_month[12] = {31, 28, 31, 30, 31, 30,
                                    31, 31, 30, 31, 30, 31};

    Drep Date::cdays[MAX_YEAR - MIN_YEAR + 1];
    int Date::init_flag = 0;

    // initialize date structures
    void Date::init_date()
    {
        int i;
        Drep cumul = 0;

        init_flag = 1;
        for (i = MIN_YEAR; i <= MAX_YEAR; i++) {
            cumul += 365 + isleap(i);
            cdays[i - MIN_YEAR] = cumul;
        }
    }

    // a leap year?
    int Date::isleap(int year)
    {
        if (year % 4)
            return 0;
        if (year % 100)
            return 1;
        if (year % 400)
            return 0;
```

```
        return 1;
}

// convert m/d/y to internal date
Drep Date::mdy_to_d(int month, int day, int year)
{
        int i;
        Drep d;

        assert(month >= 1 && month <= 12);
        assert(day >= 1);
        assert(year >= MIN_YEAR && year <= MAX_YEAR);
        assert(day <= days_in_month[month - 1] ||
           (day == 29 && month == 2 && isleap(year)));

        if (!init_flag)
                init_date();

        d = (year > MIN_YEAR ? cdays[year - MIN_YEAR - 1] : 0);
        for (i = 1; i < month; i++)
                d += days_in_month[i - 1] + (i == 2 && isleap(year));
        d += day;

        return d;
}

// convert internal date to m/d/y
void Date::d_to_mdy(Drep d, int& month, int& day, int& year)
{
        int i;
        Drep t;

        if (!init_flag)
                init_date();

        for (i = MIN_YEAR; i <= MAX_YEAR; i++) {
                if (d <= cdays[i - MIN_YEAR])
                        break;
        }
        assert(i <= MAX_YEAR);
        if (i > MIN_YEAR)
                d -= cdays[i - MIN_YEAR - 1];
        year = i;

        for (i = 1; i <= 12; i++) {
                if (d <= (t = days_in_month[i - 1] +
                   (i == 2 && isleap(year))))
```

```
                    break;
                d -= t;
        }
        assert(i <= 12);
        month = i;

        day = d;
}

// constructor from a Drep
Date::Date(Drep dt)
{
        assert(dt <= MAX_DAY);

        d = dt;
}

#define ISDEL(c) ((c) == ',' || (c) == '-' || (c) == '/')

static const char* mon[12] = {
        "jan",
        "feb",
        "mar",
        "apr",
        "may",
        "jun",
        "jul",
        "aug",
        "sep",
        "oct",
        "nov",
        "dec"
};

// constructor from a char* string
Date::Date(const char* s)
{
        char buf[3][25];
        int i;
        int j;
        int mo;
        int dy;
        int yr;

        assert(s && *s);

        // break into fields
```

```
i = 0;
for (;;) {
	if (i == 3)
		break;
	while (*s && (*s <= ' ' || ISDEL(*s)))
		s++;
	if (!*s)
		break;
	j = 0;
	if (isdigit(*s)) {
		while (isdigit(*s))
			buf[i][j++] = *s++;
		buf[i][j] = 0;
		i++;
	}
	else if (isalpha(*s)) {
		while (isalpha(*s))
			buf[i][j++] = tolower(*s++);
		buf[i][j] = 0;
		i++;
	}
	else {
		break;
	}
}
assert(i == 3);

// month

i = 0;
if (isalpha(buf[1][0]))
	i = 1;
if (isalpha(buf[i][0])) {
	if (buf[i][3])
		buf[i][3] = 0;
	for (j = 0; j < 12; j++) {
		if (!strcmp(buf[i], mon[j]))
			break;
	}
	j++;
	mo = j;
}
else {
	mo = atoi(buf[i]);
}
```

```
        // day

        i = !i;
        dy = atoi(buf[i]);

        // year

        yr = atoi(buf[2]);
        if (yr < 100)
                yr += 1900;

        d = mdy_to_d(mo, dy, yr);
}

// copy constructor
Date::Date(const Date& x)
{
        d = x.d;
}

// constructor from m/d/y
Date::Date(int month, int day, int year)
{
        d = mdy_to_d(month, day, year);
}

// conversion operator to Drep
Date::operator Drep()
{
        return d;
}

// print a date
void Date::print(char* s)
{
        int month;
        int day;
        int year;
        char buf[25];
        char* t;

        d_to_mdy(d, month, day, year);
        t = (s ? s : buf);
        sprintf(t, "%02d/%02d/%4d", month, day, year);
        if (!s)
                printf("%s", t);
}
```

```
// get m/d/y from internal
void Date::get_mdy(int& mo, int& dy, int& yr)
{
	d_to_mdy(d, mo, dy, yr);
}

// difference in days of two dates
long Date::operator-(const Date& dt)
{
	return long(d) - long(dt.d);
}

// day of week
int Date::dow()
{
	Drep dw = (d - 1) % 7 + DOW_MIN;
	if (dw > 7)
		dw -= 7;
	return dw;
}

// working days between two dates
long Date::wdays(const Date& dt)
{
	long n;
	Drep i;
	int mo, dy, yr;
	int dw;

	assert(d <= dt.d);

	n = 0;
	for (i = d; i <= dt.d; i++) {
		Date x(i);
		dw = x.dow();
		if (dw == 1)			// sunday
			continue;
		if (dw == 7)			// saturday
			continue;
		x.get_mdy(mo, dy, yr);
		if (mo == 5 && dy >= 25 && dw == 2)	// memorial
			continue;
		if (mo == 9 && dy <= 7 && dw == 2)	// labor
			continue;
		if (mo == 11 && dy >= 22 &&
		   dy <= 28 && dw == 5)		// thanks
```

```
                continue;
            if (mo == 1 && dy == 1)             // new years
                continue;
            if (mo == 12 && dy == 31 && dw == 6)
                continue;
            if (mo == 1 && dy == 2 && dw == 2)
                continue;
            if (mo == 7 && dy == 4)             // 4th july
                continue;
            if (mo == 7 && dy == 3 && dw == 6)
                continue;
            if (mo == 7 && dy == 5 && dw == 2)
                continue;
            if (mo == 12 && dy == 25)            // christmas
                continue;
            if (mo == 12 && dy == 24 && dw == 6)
                continue;
            if (mo == 12 && dy == 26 && dw == 2)
                continue;
            n++;
        }

        return n;
    }

    #ifdef DRIVER
    int main()
    {
        char buf[25];

        for (;;) {
            printf("date 1: ");
            gets(buf);
            Date d1(buf);
            printf("date 2: ");
            gets(buf);
            Date d2(buf);
            printf("calendar days = %ld\n", d2 - d1);
            printf("work days = %ld\n\n", d1.wdays(d2));
        }

        return 0;
    }
    #endif
```

1. This class represents calendar dates for the years 1875 to 2025. An actual date is stored as an absolute day number with January 1, 1875 as the basis. There are other ways of storing dates, for example by representing the month/day/year as integers.

2. The header file uses an include guard __DATE_H__ so that it can be included multiple times without error. It's common in large programming projects to have headers included more than once.
3. The Date class uses a set of private static utility functions, for example one that determines if a given year is a leap year or not. These functions are private to the class but do not operate on object instances of the class.
4. There are a set of constructors used to build Date objects. One of these is a copy constructor and two others are used to create Date objects from a month/day/year set of numbers, or from a string which has the date formatted in one of several forms:

    September 25, 1956

    9/25/56

    9 25 56

This particular constructor will be confused by dates written in the
European format, for example:

    25/9/56

5. There are member functions for determining what day of the week a given date is (Sunday - Saturday), and for computing the number of days between two dates.
6. There is also a member function for computing the number of work days between two dates (inclusive of beginning and end dates). This function is somewhat arbitrary and encodes rules used in the United States, including boundary holidays (for example, if New Year's is on a Sunday, Monday will be taken as a holiday).
7. The functions for turning month/day/year into an internal number, and vice versa, use a precomputed vector that gives the cumulative days since 1875 for a given year. Given this vector, the approach is straightforward and brute force.
8. The day of week calculation uses modulo arithmetic, based on a known day of week for January 1, 1875.
9. There are various other ways of handling dates. For example, the UNIX system represents time as the number of seconds since midnight UTC on January 1, 1970. For file timestamps and so on, a date system with a granularity of a whole day would not work. As another example, the western world changed its calendar system in September of 1752, and the above code would not work across this boundary, even if the Drep representation would handle the number of days involved.

Boyer-Moore-Horspool String Searching

As another example of how a class can represent an abstraction, consider the problem of searching a char* string for a given pattern. In the C library there is a strstr() function for doing this. But suppose that we wish to implement our own scheme, based on one of the

relatively new high-performance algorithms for searching like the Boyer-Moore-Horspool one (see the book "Information Retrieval" by William Frakes for a description of this algorithm). This particular algorithm does some preprocessing of the pattern, as a means of determining how far to skip ahead in the search text if an initial match attempt fails. The results of the preprocessing are saved in a vector, that is used during the search process.
It is quite possible but inconvenient and inefficient to code this algorithm in C, especially if the same pattern is to be applied to a large body of text. If coding in C, the preprocessing would have to be done each time, or else saved in an auxiliary structure that is passed to the search function.
But with C++, using a class abstraction, this algorithm can be implemented quite neatly:

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

class Search {
    static const int MAXCHAR = 256;
    int d[MAXCHAR];
    int m;
    char* patt;
public:
    Search(char*);
    int find(char*);
};

Search::Search(char* p)
{
    assert(p);

    patt = p;
    m = strlen(patt);

    int k = 0;

    for (k = 0; k < MAXCHAR; k++)
        d[k] = m;

    for (k = 0; k < m - 1; k++)
        d[patt[k]] = m - k - 1;
}

int Search::find(char* text)
{
    assert(text);

    int n = strlen(text);
    if (m > n)
        return -1;
```

```
        int k = m - 1;

        while (k < n) {
            int j = m - 1;
            int i = k;
            while (j >= 0 && text[i] == patt[j]) {
                j--;
                i--;
            }
            if (j == -1)
                return i + 1;
            k += d[text[k]];
        }

        return -1;
    }

    #ifdef DRIVER
    int main(int argc, char* argv[])
    {
        assert(argc == 3);

        const int MAXLINE = 256;
        char fbuf[MAXLINE];
        Search patt(argv[1]);
        FILE* fp = fopen(argv[2], "r");
        assert(fp);
        int nf = 0;

        while (fgets(fbuf, MAXLINE, fp) != NULL) {
            if (patt.find(fbuf) != -1) {
                fputs(fbuf, stdout);
                nf++;
            }
        }

        fclose(fp);

        return !nf;
    }
    #endif
```

We've added a short driver program, to produce a search program something like the UNIX "fgrep" tool.

We construct a Search object based on a pattern, and then apply that pattern to successive lines of text. Search::find() returns -1 if the pattern is not found, else the starting index >= 0 in the text.

Whether this algorithm will be faster than that available on your local system depends on several factors. A standard library function like strstr() may be coded in assembly language. Also, there's another class of string matching algorithms based on regular expressions and finite state machines, with different performance characteristics.
This simple program illustrates a way of wrapping the details of a particular algorithm into a neat package, hidden from the user.

Book Review - Inner Loops

"Inner Loops" is the name of a book by Rick Booth, published in 1997 by Addison-Wesley. It's about 350 pages, comes with a CD-ROM, and costs around $35.
The book is about performance, and has the subtitle "A Sourcebook For Fast 32-bit Software Development". It covers areas like searching and sorting, JPEG compression, matrix multiplication, and random numbers. The book has a lot of information about assembly language tricks, and also about tuning performance of C code.
It's very much a "hands on" book, and a good source of information if you are interested in pulling out all the stops to create efficient code.

Notes From ANSI/ISO

- String Literal Types
- Extern Inlines By Default
- Template Compilation Model Part 1
- Template Compilation Model Part 2
- Function Lookup in Namespaces
- Recent Changes to terminate() and unexpected
- More on terminate() and unexpected()
- Follow-up on Placement New/Delete
- Current Draft Standard Now Publicly Available

- Clarifications on Exception Handling
- the ptrdiff_t kludge for operator[]
- Return Void
- Template Default Arguments
- Resolution of Template Default Arguments
- Resolution of Return Void
- State of the C++ Standard
- Template Separate Compilation and Specialization
- The C++ Standard Library and Reserved Names
- The C++ Programming Language - Third Edition
- A Sharp Angle On Function Pointers
- State of the C++ Standard - It's Done!
- Exception Safety in Containers, Part 1
- Exception Safety in Containers, Part 2
- auto_ptr
- C++ and Signal Handling
- The Vector Constructor Ambiguity Problem
- Removal of Error-Prone Default Arguments
- Typename Changes

String Literal Types

Jonathan Schilling, jls@sco.com
[Note: this is the first of a series of columns about the details of the ANSI/ISO C++ standardization process. Jonathan Schilling works for SCO in New Jersey and is a member of the ANSI/ISO C++ committee. You should not assume that features described in this column

are available in your local C++ compiler. There is often a lag of a year or more between feature standardization and that feature showing up in an actual compiler].
At the most recent ANSI/ISO C++ standards meeting in Stockholm in July, a major change was made to the type of string literals. Previously, string literals were of type char[]; now they are of type const char[].
This repairs a longstanding blemish in C++'s type system. However, it has the potential of breaking a lot of existing code. To lessen the impact, a new standard conversion has been added to the language, from string literal to char*. (The type of wide string literals has also changed, and a similar standard conversion has been added for them).
The result is that some old code will continue to work, but some won't. For example:

```
char* p = "abc";              // used to compile; still does

char* q = expr ? "abc" : "de";  // used to compile, now an error

void f(char*);
f("abc");                     // used to compile, still does

void g(char*, int);
void g(const char*, long);
g("abc", 7);                  // used to compile, now ambiguous

template <class T> void h(T);
template<> void h<char*>(char*);
h("abc");                     // used to call specialization,
                              // now calls general template

try {
    throw "abc";
}
catch (char*) {}              // used to catch, now doesn't
```

The new standard conversion is immediately deprecated, meaning that it may be removed from the next revision of the standard. If that happens, the first and third examples above will become compilation errors as well.
One possibly confusing thing about this new standard conversion is that it operates upon a subset of values of a type (literal constants), rather than on all values of a type (which is more common). There is precedent, however, in existing standard conversions defined for the null pointer constant.
If you want to write code that will work under both the old and new rules, you can use just the new type in some contexts:

```
const char* p = "abc";
const char* q = expr ? "abc" : "de";
```

but in some contexts requiring exact type match both types must be specified:

```
try {
    throw "abc";
}
catch (char*) { /* do something */ }
catch (const char*) { /* do the same thing */ }
```

Changing the type of string literals is a big change in the language, which also introduces a significant new incompatibility with C. Whether the gain is worth the pain is a matter of opinion, but the ANSI vote was 80% in favor and the ISO vote was unanimous. It is expected that compiler vendors will provide a compatibility switch that gives string literals their old type.

Extern Inlines By Default

Jonathan Schilling, jls@sco.com

In the original C++ language, inline functions always had internal linkage. Then a change was made to allow the possibility of external linkage. Now a change has been made to make external linkage the default. This may alter the behavior of some existing code.

The linkage of a global inline function doesn't really matter if it is in fact inlined (unless it contains static local variables; see below), but in real life the inline specifier is sometimes not honored for one reason or another (see C++ Newsletter #007). When this happens, linkage matters. Consider the following:

file1.C:

```
inline int f(int x, int y) {
    return 2 * x - y;
}

... f(3, 5) ...
```

file2.C:

```
inline int f(int x, int y) {
    return 3 * x - y;
}

... f(3, 5) ...
```

That the two definitions of f() are different may be an accident or may be an intentional (but probably poor) coding practice. If the compiler does inline the function calls, the call to f() in file1 will return 1, while the apparently identical call to f() in file2 will return 4. If the functions are for some reason not inlined, but inline functions have internal linkage, then the f() calls will still return 1 and 4. This is because the compiler will generate a separate, local function body in each object file that has a call to the function, using the definition that is available for that compilation; no language rules violation occurs.

In 1994 the C++ ANSI/ISO committee made a change to the language to allow inline functions to be declared with the "extern" specifier, giving such functions external linkage

```
extern inline int f(int x, int y) { ... }
```

but the default for global inline functions was still kept as internal. (Inline member functions get the linkage of the class they are declared in, which usually means external).

The meaning of "extern inline" is that if calls to a function are not generated inline, then a compiler should make just one copy of the definition of the function, to be shared across all

object files. For example, if code in more than one object file takes the address of the above function f() and prints it out, the same address value should appear each time. Otherwise, extern inline functions are like static inline functions: the function definition is compiled multiple times, once for each source file that calls it.
Implementing this sharing requires additional linker support on some platforms, which may be part of the reason why "extern inline" is not yet supported in some C++ compilers.
Additionally, this sharing does not always have to be done; if an inline function does not contain static local variables or have its address taken, there is no way to tell whether the definition is shared or not, and a compiler is free to not share it (at the cost of increasing program size). And there is even some compiler sleight-of-hand that can avoid sharing when these conditions are present.
But what happens if, as in the original example, an extern inline function that is not inlined has different definitions in the different places it is used?
In this case, there is a violation of C++'s One Definition Rule. This means that the program's behavior is considered undefined according to the language standard, but that neither the compiler nor the linker is required to give a diagnostic message. In practice, this means that, depending on how the implementation works, the compiler or linker may just silently pick one of the definitions to be used everywhere. For the above example, both of the calls to f() might return 1 or both might return 4.
But the 1994 change did not risk altering the meaning of any existing code, because an extern specifier would have to be added to the source to trigger these new semantics.
However, at the most recent standards meeting in Stockholm in July, a further change was made to make external linkage the default for non-member inline functions. (The immediate motivation for this change was a need of the new template compilation model that was adopted at the same meeting; but more generally it was felt that changing the default was an idea whose time had come, and the change was approved unanimously in both ANSI and ISO).
With this latest change, all non-member inline functions that do not explicitly specify "static" will become external, and thus it is possible that existing code will now function differently. To help cope with this, compilers may provide a compatibility option to give inline functions their old linkage. It is also possible for users to force the old behavior by use of the preprocessor

```
#define inline static inline
```

but this only works if there are no member functions declared with "inline", and as a preprocessor-based solution is not recommended.
Note that change of behavior may occur even when there is a single source definition for a function. For example, assume that the following function is defined in a header file somewhere:
file3.h:

```
inline int g(int x, int y)
{
#ifndef NDEBUG
    cerr << "I'm in g()" << endl;
#endif
    if (x >= y)
        return h(x, y);
    else
```

```
            return 2 * x - y;
    }
```

Even though the source for the function is defined only once, the function can have different semantics depending upon where it is compiled. For example, in one file NDEBUG might be defined, but in another not. Or, the call to function h() might be overloaded and resolve differently in one file from another, depending upon what other functions were visible in each file. These cases are also violations of the One Definition Rule, and may lead to a change in behavior of existing code.

Still another way that existing inline function code can have its behavior altered is if it uses local static variables. Consider the following function e() defined in a header file:

```
inline int e() {
    static int i = 0;
    return ++i;
}
```

When the function previously had internal linkage, there was a separate "i" allocated within each object file that had a call to e(). But now that the function gets external linkage, there is only one copy of e(), and only one "i". This will cause calls to the function to return different values than before.

The One Definition Rule is a weakness of C++ where software reliability is concerned; languages with stronger module systems (such as Ada or Java(tm)) do not have these kinds of problems. As a general guideline, global inline functions should operate upon their arguments, and avoid static variables, interactions with the surrounding context, and the preprocessor.

Template Compilation Model Part 1

Jonathan Schilling, jls@sco.com

From the time templates were first introduced to C++, a problem area has been defining how templates are compiled at the source level. At the most recent C++ standards meeting in Stockholm in July, a full specification of this was made for the first time.

The crux of the issue is whether template function definitions (regular functions or member functions) are compiled separately, or must be visible within the translation units containing instantiations. Consider first the most basic source arrangement (throughout, .h and .C are

used to represent header file and source file extensions, but they may be different on any given system):

file1.h:

```
template <class T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

caller.C:

```
#include "file1.h"
void c(float x, float y) {
    float z = max(x, y);
    ...
}
```

The template function definition is included in the header file that declares the function. This is the simplest method, and up to now has been the only fully portable method; the original Standard Template Library implementation used this technique almost exclusively.

However, there is a natural reluctance to have all implementation code in header files, and so the next simplest arrangement is to move the template definitions to regular source files, and have the header pull them in:

file2.h:

```
template <class T> T max(T a, T b);
#include "file2.C"
```

file2.C:

```
template <class T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

where caller.C is the same as before (except for including file2.h rather than file1.h). The use of a regular source file for the template definition here is mostly an illusion, since file2.C is never compiled by itself but rather as part of the compilation of caller.C. But it does at least suggest a separation of interface and implementation.

A variation on this scheme that is used in some compilers permits you to leave out the explicit #include in the header:

file2a.h:

```
template <class T> T max(T a, T b);
```

with file2.C and caller.C the same as before. Here, the compiler implicitly knows by some rule where to find the corresponding .C file (usually it looks in the same directory as the .h, for a .C file with the same base name), and pulls it into the translation unit being compiled. But again, the .C file is not itself compiled.

All of these methods belong to the "inclusion" model of template compilation. It is the model that almost all current C++ compilers provide. It is relatively simple to implement and simple to understand, but while it has sufficed in practice, there are some serious flaws with it. Most of these are due to the template definition code getting introduced into the instantiating context, with unexpected name leakage as a result. Consider the following example:

file3.h:

```
template <class T> void f(T);
#include "file3.C"
```

file3.C:

```
void g(int);
template <class T> void f(T t) {
    g(0);
}
```

caller3.C:

```
#include "file3.h"
void g(long);
void h() {
    f(3.14);
    g(1);          // hijacked!
}
```

Clearly the writer of caller3.C expected the g(1) call to refer to the g(long) in the same source file. But instead, the g(int) in file3.C is visible as well, and is a better match on overloading resolution. While use of namespaces can alleviate some of these problems, similar things can happen due to macros:

caller3a.C:

```
#define g some_other_name
#include "file3.h"
void h() {
    f(3.14);
}
```

This time, the call g(0) in file3.C, which is clearly intended to refer to the g(int) in that file, gets altered by the macro defined in the context of caller3a.C.

None of these problems would occur if file3.C were separately compiled, because there would be no possibility of its context and the caller's context becoming intermingled in unexpected ways. While these kinds of context problems can also occur in inline functions (see C++ Newsletter #015), the potential for damage with templates is much greater, given the centrality of templates to modern C++ libraries and applications.

A separate compilation model for templates was envisioned as part of the C++ language template design from the start, but was never specified in any detailed way. The first attempt to (partially) implement it (Cfront 3.0) ran into difficulties, and subsequently compiler vendors shied away from it. The first attempts to specify it in the draft ANSI/ISO standard were criticized as poorly specified, hard to use, and hard to implement efficiently. A series of contentious discussions and reversals ensued, but now by way of invention and compromise, a (what is hoped to be) clear and reasonably efficient version of separate compilation has been made. In addition, the de facto existing "inclusion" model is also permitted by the standard (but the implicit inclusion method, illustrated by file2a.h above, will not be, unless by vendor extension).

Template Compilation Model Part 2

Jonathan Schilling, jls@sco.com

In the last issue, we looked at the "inclusion" model of template compilation, which is the one used by most compilers in practice but which is lacking in several respects. As a reminder, here was the example that illustrated name leakage in the inclusion model:

file3.h:

```
template <class T> void f(T);
#include "file3.C"
```

file3.C:

```
void g(int);
template <class T> void f(T t) {
    g(0);
}
```

caller3.C:

```
#include "file3.h"
void g(long);
void h() {
    f(3.14);
    g(1);   // should call g(long), but calls g(int) instead
}
```

Now we'll look at the newly-specified template separate compilation model that has recently been added to the standard. There isn't space here to go into a full description of the new rules, and in fact the complexity of this subject rapidly approaches infinity! But here are some of the key highlights:

Names in template functions are divided into those that are dependent upon the template arguments, and those that are not. This distinction is made syntactically, making it easier for people and compilers to understand.

Names in template functions that are not dependent upon the template arguments are resolved only in the template definition context (an example would be g(0) in file3.C above).

Names in template functions that are dependent upon the template arguments are resolved either in the template instantiation context (using external names that may be found in object code symbol tables) or in the template definition context. In the case of nested or transitive instantiations, no "intermediate context" is available.

Instantiation of template functions is made "position independent", meaning that if the meaning of a program changes depending upon where instantiations are placed, program behavior is undefined.

Instantiations may be performed at either compile- or link-time. If the choice makes a difference, program behavior is undefined. An implementation is allowed to place compilation-order

restrictions on separately-compiled templates.

Separate compilation of templates is not done by default: the template declaration or definition must use the new keyword "export" in order for it to happen. Otherwise the inclusion method is used. This will provide upward compatibility of existing template code.

Some of these changes involve the template instantiation model (see C++ Newsletter #010) more than the template source model, but are necessary to make separate compilation workable.

Here's the example from above, made into a separately-compiled template:

file4.h:

```
template <class T> void f(T);
```

file4.C:

```
void g(int);
export template <class T> void f(T t) {
    g(0);
}
```

caller4.C:

```
#include "file4.h"
void g(long);
void h() {
    f(3.14);
    g(1);   // now, this calls g(long); g(int) not visible
}
```

In this model, file4.C is compiled explicitly, as well as caller4.C, and its source is not pulled into the header, explicitly or implicitly. The source is otherwise identical to the inclusion model except for the addition of the "export" keyword. (The meaning of this keyword is somewhat similar to the existing "extern" keyword, and some people wanted to reuse that keyword rather than introduce a new one. After some debate, the committee decided at its recently concluded November meeting not to overload "extern". Also note that the keyword may be placed on either the template declaration or the template definition; this flexibility may help library vendors in shipping products that can be used with either template compilation model).

One area of uncertainty is how much the "no intermediate context" limitation will affect real code. Here's an example where it matters:

ic1.h:

```
export template <class T>
void g(const T&);
```

ic1.C:

```
export template <class T>
void g(const T& t)
{
    length(t);      // how does length get found?
}
```

ic2.h:

```
export template <class T> void f(T);
```

ic2.C:

```
#include "ic1.h"

template <class T>
class Container { ... };

export template <class T>
int length (const Container<T>&) { ... }

export template <class T> void f(T t)
{
    Container<T> s;
    g(s);
}
```

ic3.C:

```
#include "ic2.h"

class A { ... };

void m() {
    A a;
    f(a);       // this starts the instantiations
}
```

This is a case of transitive instantiation, where m() instantiates f(A) which instantiates g(Container<A>). Within g(), length(t) is a dependent name lookup, so it can find length either in the definition context (ic1.C) or in the instantiation context (ic3.C). But it's in neither. It's in ic2.C, which is considered "intermediate context". Thus this example would not compile as is, and would have to be recoded to use the inclusion method (basically, drop the "export"'s and include the .C's into the .h's).

It is an open question how common this kind of intermediate context problem will be. One analysis found no cases of it in the template-intensive Standard Template Library, which may be encouraging. As with many of the new inventions of the C++ standardization process, only time will tell.

Function Lookup in Namespaces

Jonathan Schilling, jls@sco.com

An important change has recently been made in the way functions are found within namespaces.

The three basic ways of making the contents of a namespace visible were discussed in C++ Newsletters #002 and #004. These are: explicit qualification, using directives, and using declarations.

Consider the following namespace, which declares a class and some (non-member) functions:

```
namespace N {
    class A { ... };
    A& operator+(const A&, const A&);
    void f(A);
    void g();
}
```

Now consider the following function that takes arguments of the class type:

```
void z(N::A x, N::A y)
{
    x + y;      // (1)
    f(x);       // (2)
    g();        // (3)
}
```

Given the original rules for namespaces (just the three basic methods of namespace visibility), all three of the statements in this function are compilation errors, because none of the functions being called are visible.

However the standards committee has changed the way functions are looked up. Now there is a new language rule, which says that the namespaces of the classes of the arguments, and the namespaces of the base classes of those classes, are included in the search for function declarations, even when the contents of those namespaces are not otherwise visible.

So, when looking for an operator+() in (1) above, the arguments are x and y, the class of those arguments is A, and A is declared in namespace N. Thus the compiler looks for an operator+() in N, and finds one, and the call is legal. A similar process happens for the call to f() in (2). However, the call to g() in (3) is still a compilation error, because there are no arguments to direct the lookup. The call would have to be made using one of the basic methods:

```
N::g();     // explicit qualification
```

If the arguments to the function have different types, then all the associated namespaces are searched. Arguments of built-in types such as int have no associated namespace, while arguments of more complicated types such as pointers to functions bring in the namespaces of the pointed-to function's parameters and return type.

This new lookup rule was first added to solve some technical language definition problems with operator functions. It was then added to solve some other problems with template "dependent name" lookup (see C++ Newsletter #017) and template friends. At that point it was felt that consistency demanded the new rule be extended to lookup of all functions in all contexts, and this was done (albeit with some dissent within the committee) at the Stockholm meeting in July.

Because of the staggered introduction of this rule, for a while you may encounter compilers that implement it for operator functions but not for other functions, but eventually all implementations will be in full conformance.

One important thing to note about this rule change is that it is a step toward making namespaces a powerful scoping and packaging construct, rather than just a transparent vehicle to avoid name collisions. The art of employing namespaces is still in its early stages, and first reports have indicated that the basic methods of making names visible are sometimes too

verbose (explicit qualification), too broad (using directives), or too prone to error and omission (using declarations). The new rule may help alleviate some of these problems.

Recent Changes to terminate() and unexpected()

Jonathan Schilling, jls@sco.com

Earlier in this issue the basic purposes of the terminate() and unexpected() functions are described. In the past year the standards committee has made several refinements to these functions.

The committee has confirmed that direct calls may be made to these functions from application code. So for instance:

```
#include <exception>
...
    if (something_is_really_wrong)
        std::terminate();
```

This will terminate the program without unwinding the stack and destroying local (and finally static) objects. Alternatively, if you just throw an exception that doesn't get handled, it is implementation- dependent whether the stack is unwound before terminate() is called. (Most implementations will likely support a mode wherein the stack is not unwound, so that you can debug from the real point of failure).

Probably the main purpose of making direct calls to terminate() and unexpected() will be to simulate possible error conditions in application testing, especially when the application has established its own terminate and unexpected handlers.

The committee has changed slightly the definition of what handlers are used when terminate() or unexpected() are called. In most cases, they are now the handlers in effect at the time of the throw, which are not necessarily the current handlers. Usually they are one and the same, but consider:

```
#include <exception>

void u1() { ... }
void u2() { ... }

class A {
public:
    A() { ... }
    A(const A&) { ... std::set_unexpected(u2); }
};

void f() throw(int)
{
```

```
        A a;
        throw a;        // which unexpected handler gets called?
}

int main()
{
        std::set_unexpected(u1);
        f();
        return 0;
}
```

The copy constructor for A is called as part of the throw operation in f(), so by the time the C++ implementation determines that an unexpected handler needs to be called, u2() is the current handler. However, based on this recent change, it is the handler in effect at the time of the throw - u1() - which gets called. On the other hand, if a direct call to terminate() or unexpected() is made from the application, it is always the current handler which gets called. Some would argue that this kind of rule just adds complexity without much benefit to already-complex C++ implementations, but others feel that if an application is going to be dynamically changing its terminate and unexpected handlers, retaining the correct association is important. In the next issue we'll talk about another clarification of terminate() and unexpected(), this time related to the uncaught_exception() library function introduced above.

More on terminate() and unexpected()

Jonathan Schilling, jls@sco.com

In C++ Newsletter #019 the terminate handler, the unexpected handler, and the standard library function uncaught_exception() were introduced.

The standards committee recently decided what values uncaught_exception() should return when called from these handlers: false from unexpected() and true from terminate().

The latter ruling is somewhat counter-intuitive, because an exception is considered "caught" in the standard when terminate() is called, so logically uncaught_exception() should return the inverse. The rationale for the decision was that uncaught_exception() should include the case where terminate has been called by the implementation. Some committee members argued that it should return false, or that the value should be left undefined. But at the end of the day this is a good example of the kind of minutiae a standards committee must deal with, because if you consider that the purpose of uncaught_exception() is to help keep you out of terminate(), then if you're already in terminate() anyway it pretty much doesn't much matter what it returns.

Note however that these rules only apply when unexpected() and terminate() are called by the implementation. When direct user calls are made to these functions (see again Newsletter #019), uncaught_exception() will return false unless the direct user call was made from code executing as part of an exception. In the case of terminate() this difference between implementation calls and direct calls might complicate simulation testing of error conditions.

Follow-up on Placement New/Delete

Jonathan Schilling, jls@sco.com
Also introduced in Newsletter #019 were placement new and placement delete. In addition to the language providing this general capability, the C++ standard library also provides a specific instance for void*:

```
void* operator new(size_t, void*);

void operator delete(void*, void*);
```

These are accessed by saying:

```
#include <new>
```

These functions are defined to do nothing (though new returns its argument). Their purpose is to allow construction of an object at a specific address, which is often useful in embedded systems and other low-level applications:

```
const unsigned long MEMORY_MAP_IO_AREA = 0xf008;

...

Some_Class* p = new ((void*) MEMORY_MAP_IO_AREA) Some_Class();
```

Based on a fairly recent decision of the standards committee, this definition of placement new/delete for void* is reserved by the library, and cannot be replaced by the user (unlike the normal global operator new, which can be). The library also defines a similar placement new/delete for allocating arrays at a specific address.

Current Draft Standard Now Publicly Available

Jonathan Schilling, jls@sco.com
In C++ Newsletter #018 it was mentioned that the C++ standards committee has recently issued its "second Committee Draft" (CD2) of the standard, with an associated public review period, but that due to ISO policy the draft would not be available without charge.
ISO has just now reversed this policy, and two Web sites now have information on how to download the draft and to make public review comments:

http://www.setech.com/x3.html

http://www.maths.warwick.ac.uk/c++/pub/

The ANSI public review period ends on March 18, so if you're interested in submitting a comment, better do it quickly!

Clarifications on Exception Handling

Jonathan Schilling, jls@sco.com

The ANSI/ISO C++ standards meeting earlier this month in Nashua, New Hampshire, produced some clarifications of exception handling semantics.

One interesting case is this one, which was featured in the January 1997 issue of the magazine C++ Report:

```
try {
    // exception prone code here, that may do a throw
}
catch (...) {
    // common error code here

    try {
        throw;  // re-throw to more specific handler
    }
    catch (ExceptA&) {
        // handle ExceptA here
    }
    catch (ExceptB&) {
        // handle ExceptB here
    }
    catch (...) {
        // handle unknown exceptions here
    }

    throw;
}
```

The idea behind the code is to factor out common error handling logic into the first part of the catch handler (so as not to replicate it), rethrow the exception to get error handling specific to the exception in the individual inner handlers, and then finally to rethrow the exception again to let functions further up the call chain do their handling.

The question is, does this code work as intended? The draft standard speaks of a throw creating a temporary object that is then deleted when the corresponding handler exits. Does this mean that when the inner handlers above exit, the rethrow will be of a nonexistent temporary object? The standard isn't really clear on this, and some existing compilers have been found to do the deletion at the inner handler, with the result that the program crashes.

The answer is that this code should indeed work as intended, and that the existing compilers for which this does not work are wrong. (Fortunately SCO's new C++ compiler is one of the ones that is getting it right!).
Furthermore, the committee stated that the value of the standard library function uncaught_exception() (see C++ Newsletter #019) changes (from false to true) at both of the rethrows, until such time as the rethrown exception is caught again.
Another exception handling issue that was clarified is whether base class destructors are called when a derived class destructor throws an exception:

```
class B {
public:
    ~B() { ... }
};

class D : public B {
public:
    ~D() { throw "error"; }
};

void f() {
    try {
        D d;
    }
    catch (...) { }
}
```

Does ~B() get called as well as ~D()? The answer is yes. This may seem almost obvious -- it is part of the general principle of C++ that constructed subobjects always get destroyed if something goes wrong with the enclosing object -- but in fact there was some debate on this within the committee.
Finally, one of the comments from the ANSI public review period concerned an area of exception handling that needed no clarification but is often misunderstood:

```
try {
    throw 0;
}
catch (void *) {
    // does the exception get caught here?
}
```

The handler should not catch the exception, but apparently in some compilers it does. The draft standard is clear that throw and catch types either have to match exactly, or be related by inheritance, or be subject to a pointer-to-pointer standard conversion. Since 0 is not of a pointer type, the last requirement isn't met, and no handler is found. Similarly note that the whole range of other standard conversions do not apply, so that for example a handler of type long does not catch an exception of type int.

the ptrdiff_t kludge for operator[]

Jonathan Schilling, jls@sco.com

Consider the following innocent-looking fragment of a C++ string class:

```
class String {
public:
    String();
    char& operator[](unsigned int);
    operator const char*();
};
```

The class has an indexing operator to get at characters of the String, and there's also a conversion operator to convert the String into a C-level const char* string. A typical usage might be:

```
String s;
char c = s[0];
```

But there is a potential ambiguity here. Is the expression s[0] equivalent to:

```
s.operator[](0)
```

or:

```
((const char*)s)[0]
```

The second alternative is more strictly:

```
(s.operator const char*())[0]
```

The intuitive answer is the first alternative, and for a long time compilers interpreted it as such. This was relying on a nonstandard weighting of operator[] in overload resolution, however, and beginning two or three years ago compilers began implementing this according to the draft ANSI/ISO standard, in which it was ambiguous and an error.

It was ambiguous because the first interpretation involves an exact match (String s, which is implicitly an argument to the member function) and a (standard) conversion (the literal 0, which is of int type, to unsigned int), while the second interpretation also involves an exact match (int 0 to the built-in [] operator) and a (user-defined) conversion (the String s to a const char*). Neither interpretation wins out.

Flagging this ambiguity broke a lot of existing code, including commercial library string classes from Rogue Wave's Tools.h++ and USL C++ Standard Components, as well as application code.

The ways around this ambiguity were to either (1) add an overloaded operator[](int) to the class, (2) get rid of the conversion operator, or (3) make all indexing expressions of unsigned int type.

The first approach isn't great because it adds clutter (for a set of values - negative integers - rarely used in indexing) but is the least disruptive remedy. The second approach may be desirable for other reasons (conversion operators often cause more trouble than they're worth) and is what was done in the new ANSI/ISO standard library string class, but will likely require users of existing classes to modify their code. However if you don't have control over the class's source, the third choice must be taken. Changing the indexing type can be an improvement in some contexts:

```
unsigned int i; // not int i
s[i];
```

but is real ugly in others:

```
s[0u];        // not s[0]
```

Since this has been one of the C++ usage problems most commonly reported to compiler vendors, the ANSI/ISO C++ standards committee tried to improve things a bit last year. It changed the "pseudo-prototype" of the built-in operator[] to take a parameter of type ptrdiff_t (the predefined integer type guaranteed to hold pointer differences, as defined in header <cstddef>) rather than int.

This has two effects. It provides ptrdiff_t as a unique type that user-defined overloaded operator[] functions can be written to, so that if you recode the class to use ptrdiff_t as the parameter type to operator[]:

```
char& operator[](ptrdiff_t);
```

then expressions such as s[0] will no longer be ambiguous. This is because the int 0 will be the same (either an exact match or a standard conversion) for both interpretations, and thus the s.operator[](0) interpretation will win out because of its exact match on String s. This does however have the drawback of making the indexing type signed, which presumably you didn't want when you made it unsigned int in the first place.

The second effect is that if your system defines ptrdiff_t to be something other than int (such as long), s[0] will no longer be ambiguous even with the original definition of the class where the indexing type is unsigned int. This is because the (s.operator const char*())[0] interpretation will involve a second conversion (int 0 to long).

The committee's action is still something of a kludge, because whether a simple usage compiles or not suddenly becomes dependent upon how an apparently unrelated system header type is defined. But it allows more C++ code to behave in an intuitive fashion, and that's always a good thing.

Return Void

Jonathan Schilling, jls@sco.com

One issue that the C++ standards committee has discussed several times is allowing the return statement to return expressions of type void. An example would be this:

```
void m();

void n() {
    return m();
}
```

Currently, this is not allowed; the proposal is to extend the language to allow a return statement to have an expression of void type, within a function of void return type.

The motivation is to make it easier to write templates. Consider this example, which in a different form appears in the new standard library:

```
template <class T>
T f(T (*pf)()) {
    ...
    return pf();
```

```
}

int g();
void h();

...

f(g);       // currently allowed
f(h);       // currently an error, proposed to be allowed
```

Without the extension to the language, a template specialization would have to be provided for the void case:

```
template<> void f(void (*pf)()) {
    ...
    (*pf)();
    return;
}
```

Once you start playing with extensions involving void, there's a temptation to go further toward making void a first-class type. Doing this might ease template writing in other contexts, but the effort quickly runs into language definitional problems. The argument for this limited extension is that it isn't really an introduction of void objects or void arguments, but rather a realization that the currently allowed:

```
return ;
```

already has an implicitly void expression in it.

While this proposal has not yet been formally adopted (the standard is currently in a review and balloting phase and actual changes to the working paper cannot be made), it received widespread support at the recent Nashua meeting and is likely to be voted in at the next meeting this summer.

Template Default Arguments

Jonathan Schilling, jls@sco.com

If you look at the currently available "second Committee Draft" of the proposed standard (see C++ Newsletter #020), you'll find some very curious wording in Clause 17.3.4.4:

> Throughout the C++ Library clauses (17 through 27), whenever a template member function is declared with one or more default arguments, this is to be understood as specifying a set of two or more overloaded template member functions. The version with the most parameters defines the interface; the versions with fewer parameters are to be understood as functions with fewer parameters, in which the corresponding default argument is substituted in-place.

This "standards rewrite rule" is an example of the last-minute standards patching that goes on when both a language and a library are being aggressively designed at the same time, and then

somebody discovers that the library depends upon a language feature that doesn't exist. Here's the problem:

```
class A {
public:
    A();      // has a default constructor
};
A a;

class B {
public:
    B(int);   // doesn't have a default constructor
};
B b(19);

template <class T>
void f(T t1, T t2 = T());      // note the default argument

void g() {
    f(a, a);   // 1) ok, both arguments present
    f(a);      // 2) ok, second argument defaults to A::A()
    f(b, b);   // 3) ???
    f(b);      // 4) error, 2nd arg defaults to absent B::B()
}
```

The issue is when and where dependent default arguments of template functions get instantiated. Currently they get instantiated in the context of the declaration of the function, which means that the line 3) call above is an error, because B::B() is looked up and found not to exist. However significant parts of the new standard library have been written under the assumption that line 3) will compile, on the grounds that the default argument is not actually needed, and that only line 4) should cause an error.

So what to do now?

The problem could be worked around in the library by adding overloaded function signatures (as in the current draft's rewrite rule) or by using an additional intermediate template, but neither solution is concise or graceful, either for the standard library or for user-written classes in the years to come.

On the other hand modifying the language to not instantiate default arguments unless needed involves the usual complexities of template instantiations and their context (see C++ Newsletter #016 and #017) and is a tricky change to make this late in the standards process.

This inconsistency between language and library arose by mutual confusion and happenstance, which of course made the discussion about it at the last meeting especially overwrought, with no consensus reached. The question will probably be decided at the standards meeting next month, and we'll let you know what happens.

Resolution of Template Default Arguments

Jonathan Schilling, jls@sco.com

In the last issue (#024) we looked at a problem where the draft standard library relied on a non-existent feature in the draft standard language. This feature was template default arguments not getting instantiated unless used.

At the meeting of the ANSI/ISO C++ standards committee just concluded in Chiswick, London, a change to the language was approved that will resolve this discrepancy.

First, to recap the problem:

```
class B {
public:
    B(int);    // class doesn't have a default constructor
};
B b(19);

template <class T>
void f(T t1, T t2 = T());       // function has default argument

void z() {
    f(b, b);   // used to be error, under new rule is ok
    f(b);      // error, both old and new rules
}
```

Under the old rules, the instantiation of f(b, b) occurred in the context of the definition of the template function, at which point the default argument value B::B() would be looked up, and would cause an error because it did not exist.

Under the new rule, a distinction is made between template arguments that depend upon the template formal parameters and those that don't. This distinction is already made in other template contexts such as name lookup and separate compilation (see issue #017) so no new specification wording is needed.

For non-dependent default arguments, name binding and error checking may be done at the point of function definition (but need not be, in which case it is done at the point of instantiation). However, for dependent default arguments, name binding and error checking may only be done when the default argument is needed, that is, when the function is called without the argument.

Thus, in the above example, the default argument is dependent (since it involves the template formal parameter T), and so the instantiation of f(b, b) does not cause any error, since the default argument is not needed. However the instantiation of f(b) would cause an error, since then the default argument is needed.

For an example of a function with a non-dependent default argument, consider this addition to the example:

```
template <class T>
void g(T t1, char c2 = b);

void y() {
    g(b, 'x');      // error, old and new rules
}
```

The default argument expression has nothing to do with the template formal parameter T, and so the error in the default expression (an object of class B cannot be converted to char) must be given, even though the default argument is not used. This error may be given either at the point of the definition of g() or at the point of instantiation.
The discussion of this issue in the committee continued to be hotter than it needed to be, but in the end there was a large majority supporting the language modification. This means that the library can remain unchanged in its use of template default arguments.

Resolution of Return Void

Jonathan Schilling, jls@sco.com
In C++ Newsletter #023 we discussed a proposed change to allow return statements to return expressions of type void. At the standards meeting just concluded, this change was approved.

State of the C++ Standard

Jonathan Schilling, jls@sco.com
At the meeting of the standards committee, national body comments concerning the second "Committee Draft" of the standard were discussed and, in many cases, addressed by incorporating changes into the current working paper.
The committee is still on schedule to produce a "Draft International Standard" in December 1997, which despite its name will essentially be the final standard (modulo corrections for grievous typos). This would then be subject to formal ballot and approval during 1998.

Template Separate Compilation and Specialization

Jonathan Schilling, jls@sco.com
In C++ Newsletter #017 the new template separate compilation model was described. Since its introduction the ANSI/ISO standards committee has been dealing with sorting out the loose ends from this addition to the language.
Take the following case, for instance:
file1.h:

```
template<class T> class A { };
```

```
    template<class T> void f(T);
file1.C:
    #include "file1.h"
    export template<class T> void f(T) { A<T> a; }
```

These files introduce a template class, and a template function whose definition depends upon the template class. The keyword "export" says that the template function definition doesn't have to be included into translation units that call it; rather it can be separately compiled.

```
file2.C:
    #include "file1.h"
    template<> class A<int> { };    // line 2
    void g() ( f(1); }              // line 3
```

Line 2 of this file declares a specialization (see C++ Newsletter #012) of the class template for type int, and then line 3 instantiates the function template for type int, which in turns instantiates the class template for type int. But the function template definition in file1.C cannot see the specialization in file2.C. So how does this work? Does the specialization not get used? Or does compilation of file1.C get deferred somehow until the specialization is seen?

At the recent standards meeting in London, the committee decided that when an explicit specialization is not visible in an instantiation context, yet would affect the instantiation (such as in this case), the program is ill-formed, but that no diagnostic is required.

This ruling falls into the same general behavior as the One Definition Rule (see C++ Newsletter #015). As such, it makes things easier for compiler implementors but harder for language users. For instance, the writer of file2.C may not look at the implementation of f(), and thus may not realize that specialization of class A will cause (possibly silent) undefined behavior.

As we've editorialized before, the One Definition Rule is a weak area of C++, especially when compared to languages such as Ada and Java(tm) which stay well-defined across separate compilation boundaries. The problems mostly result from the use of #include files as the mechanism for modularity; improving this was something occasionally discussed in C++ standardization circles but never really tackled. Maybe in C++ 0X!

The C++ Standard Library and Reserved Names

Jonathan Schilling, jls@sco.com

The draft ANSI/ISO C++ standard library incorporates by reference much of the C standard library. This has always been the case, back to the earliest days of C++; what has changed during the standardization process is the placement of C standard library names into namespace std, the namespace that also holds the C++ standard library.
Thus, a proper C++ program now calls the C standard library like this:

```
#include <cstdio>
...
std::printf("hello, old library\n");
```

rather than in either of the ways it used to:

```
#include <stdio.h>
...
::printf("hello, old library\n");       // explicit scoping
                                        //    or
printf("hello, old library\n");         // lazy but more typical
```

The old forms are still accepted by virtue of a deprecated backward compatibility provision, which states that <stdio.h> has the effect of pulling in <cstdio> and making its std:: names visible as if there were using declarations for them.
But what occurs if you have declared your own name in the global namespace, that happens to be the same as one of the names in the C standard library? That is, something like:

```
double printf = 3.1416;
```

Is this well-formed? Does it depend upon whether <cstdio> is present?
Or upon whether <stdio.h> is present?

At the most recent standards meeting in London in July, the committee decided to reserve all C standard library names for the implementation, in both namespace std and in the global namespace, regardless of whether any of the headers that define those names are present. So for example the above declaration of printf would result in undefined behavior.
This decision was made primarily to make C++ compiler and library vendors' lives a little easier, since for various reasons putting the C standard library into namespace std has proved to be a major headache for them. (Indeed some vendors have tried for several meetings to get the C standard library taken out of namespace std altogether, but this has been rejected by the committee).
With a commonly known name such as printf, it's unlikely anyone will declare it themselves. But there are many names in the C standard library, and it is possible to collide with some of the more obscure ones. The best advice is this:

Stay out of the global namespace.

That is, all C++ application code should be put into an appropriate namespace (named or unnamed; see C++ Newsletter issues #001 through #004); then you will never risk collision with implementation-reserved names or names that come in through system headers (unless they are macros, against which namespaces offer no protection).

The C++ Programming Language - Third Edition

Jonathan Schilling, jls@sco.com
Bjarne Stroustrup, the inventor of C++, has now published "The C++ Programming Language, Third Edition" (Addison-Wesley, about $43). This is a massive update of his earlier Second Edition, incorporating all of the ANSI/ISO changes to the C++ language and library.
As in the past, this book is not the gentlest introduction to the language, but is aimed for the able programmer. It retains the strength of the previous editions in emphasizing design considerations as well as language details. The new standard library is well covered, as are the motivations for using the new language features. Each chapter concludes with a concise "Advice" section that give helpful rules of thumb for language and library users. For example the recommendation above is given on p. 194 as

> Place every nonlocal name, except main(), in some namespace.

The book is quite large (900 pages), but then C++ is now a big language. It was not proofread as well as it might have been, so it is a good idea to print out the Errata at http://www.awl.com/cp/stroustrup3e. The Errata also include clarifications and updates due to late changes in the standardization process.

A Sharp Angle On Function Pointers

Jonathan Schilling, jls@sco.com
At the summer 1996 ANSI/ISO standards meeting, the concept of language linkage was extended to cover pointer to function types, such that there are now pointers to C functions and pointers to C++ functions as distinct types. This change was described in C++ Newsletter issue #021, "A New Angle on Function Pointers".
Now that this feature is being introduced into compilers, such as SCO's new C++ compiler, we are seeing how it can break existing code.
Consider first this usage:

```
something.h:
        extern "C" {
                void f(int);
        }

something.C:
        #include "something.h"

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

where both header and source file are compiled with C++ (that is, this is a C++ function that you want to be callable from C as well).
The definition in the source file does not have extern "C" on it, but because it is the same f(int) as in the declaration in the header, the language rules state that the extern "C" from the declaration applies to it as well. The definition for f is thus generated with C linkage, that is, with the unmangled name "f".

Now suppose we do the same thing, but with a pointer to function as a parameter:

something.h:

```
extern "C" {
    void g(int (*pf)());
}
```

something.C:

```
#include "something.h"

void g(int (*pf)()) { ... }
```

Under the old language rules, the definition of g would get its language linkage from the declaration of g, and thus would be generated with C linkage and the unmangled name "g", just as in the case above. But under the new language rule, g is generated with C++ linkage and a mangled name such as "g__FPFv_i", and a link-time error will likely result.

Why does this happen? Because now the language linkage applies not only to the function it is specified for, but also to any other names or declarators introduced by the declaration of that function. In this case this means the linkage applies to the pointer-to-function pf.

So the declaration of g is within an extern "C" block, which means that its pf is considered a pointer to a C function. However the definition of g is not within an extern "C" block, which means that its pf is considered a pointer to a C++ function. Thus the parameters are of different types, thus the two g's are considered different and overloaded rather than the same function, thus the extern "C" on the first g does not extend to the second, and thus g is generated with C++ linkage and a mangled name.

There are two ways the code can be revised to work properly. The first is to put the definition into an extern "C" block as well:

something.C:

```
extern "C" {
    void g(int (*pf)()) { ... }
}
```

This works because the definition's pf will now also be interpreted as a pointer to C function, and so the two g's are the same.

The second approach is to use a typedef for the pointer to function:

something.h:

```
extern "C" {
    typedef int (*pf)();
    void g(pf);
}
```

something.C:

```
void g(pf) { ... }
```

This works because the typedef preserves the "pointer to C" characteristic of the type when pf appears again in the definition of g.

Either of these approaches will also work properly with compilers implementing the old language rules.

State of the C++ Standard - It's Done!

Jonathan Schilling, jls@sco.com

The ANSI/ISO standards meeting that was held earlier this month in Morristown, New Jersey finally completed the C++ standard. Procedurally, this means that the Final Draft International Standard was produced and approved at this meeting, and will be forwarded on to the various national bodies for approval as an International Standard, with no further changes allowed. This final ratification is expected by March 1998, and the standard will become officially official some time after that. But in terms of the wording of the standard, it's done now.

The standard has taken eight years and a cast of hundreds to develop. It is significant to note that at the Morristown meeting the vote to approve the standard was unanimous, both within ANSI and ISO. Furthermore, all of the major objections that had been raised by countries that voted "No with comments" or "Yes with comments" on the CD2 ballot (see C++ Newsletter #018) have now been resolved to the satisfaction of the national bodies involved.

In other words, this is a great moment for the C++ world. We will now have one dialect of the language and library instead of many; applications will be able to be written in a portable way, and compiler and library vendors will be able to spend more resources on quality of implementation issues. The new abstraction mechanisms introduced by the standard (such as generic programming and the STL) will now become available to all of the estimated 1.5 million C++ programmers worldwide.

A press release on the completion of the standard can be found at:

http://www.research.att.com/~bs/iso_release.html

Exception Safety in Containers, Part 1

Jonathan Schilling, jls@sco.com

One of the best improvements made in the standard happened at the last two meetings in London and Morristown. This is the requirement that library containers (that is, the STL) behave in an "exception safe" manner if an exception is thrown during a container operation. (Typically such exceptions might come from a failed new or from a copy constructor for the contained object). Prior to these changes, if an exception was thrown, all bets were off, and as a consequence the library didn't work well with the language's recommended way of reporting error conditions.

What does "exception safe" mean? Basically there are three levels of guarantees:

1. Certain operations will do nothing if an exception is thrown during the operation.

2. Other operations do not have that guarantee, but at least will not leak memory, fail to destruct constructed objects, or behave in an undefined manner upon destruction of the container.

3. Destructors in the library are guaranteed not to throw an exception.

The first level of guarantee can be thought of as "commit or rollback" semantics. For example, if you insert an object into a list, you can know that if the insertion is successful the list will now contain that object, or if it is unsuccessful the list will remain unchanged.
The second level of guarantee is weaker than that. It simply states that the contents of the container are undefined, but the program will still behave reasonably otherwise. Thus, for example, if an insertion into a vector is unsuccessful, you will still have a working, destructible vector, but its contents are unknown -- it might be unchanged from before the operation, or it might be empty, or anywhere in between.

The third guarantee is simply to ensure sane behavior of the library.
As a general rule, no destructor should ever throw an exception!

The rationale for the difference between the two levels of guarantees is based on how the different STL containers are implemented and the difficulty of supporting the guarantees; this and other details of exception safety in containers will be discussed in more detail in the next issue.

Exception Safety in Containers, Part 2

Jonathan Schilling, jls@sco.com
In the previous issue we mentioned that one of the best improvements made to the standard over the last two meetings has been the requirement that standard library containers exhibit certain levels of exception safety.

Here's an example of what that means, using lists and vectors (C++ Newsletter #015):

```
#include <iostream>
#include <list>

using namespace std;

class A {
public:
    A(int i) { n = i; }
#if CCTOR
```

```
        A(const A& a) {
            if (a.n < 6)
                n = a.n;
            else
                throw "too large";
        }
    #endif
        int get() const { return n; }
    private:
        int n;
    };

    int main() {
        list<A> la;
        la.push_back(A(0));  la.push_back(A(1));

        typedef list<A>::iterator LI;

        try {
            for (int i = 2; i < 10; i++) {
                LI li = la.begin();  li++;
                la.insert(li, A(i));
            }
        } catch (const char* s) { }

        // what does la contain now?
        for (LI i = la.begin(); i != la.end(); i++)
            cout << i->get();
        cout << endl;
    }
```

The list initially has two elements, and then we insert a series of elements before the second element position. When compiled without any user-supplied copy constructor, the output is:

```
    0987654321
```

What happens when we compile with CCTOR defined? We have supplied a copy constructor which throws an exception for element 6. This copy constructor is called by the library container implementation when elements are copied into the container. What is the state of the container when the copy construction for element 6 throws an exception?

Prior to the recent changes to the standard, the results of this were completely undefined. After the exception was caught the list might be intact or not; the iterator operations on it might work or not; there might be fewer or no elements still in the list; or the destruction of the list object at the end of main() might cause a core dump.

However, now the standard states that the insert() operation that causes the throw will have no effect, and guarantees that the output will be:

```
    054321
```

which is what it was before the insert() that threw began. In other words, list insert is done with "commit or rollback" semantics.

Now, let's take the above example and use a vector rather than a list. (This can be done by simply editing the three text occurrences of "list" to "vector").
Without the copy constructor, we get the same output as for list:

0987654321

But with the copy constructor included via defining CCTOR, we get:

0543211

which looks like a "wrong" value (there's an extra 1). What happened?
Remember in the previous issue we talked about two levels of guarantees in terms of exception safety in containers. The stronger level is the commit-or-rollback level, and is required for most operations on lists, maps, and sets. The weaker level doesn't guarantee the contents of the container, but does guarantee that the container will be well-formed (for example, you can iterate through it and destruct it). This weaker level is all that is required of most operations on vectors and deques.
The basic rationale for the difference is to permit efficient implementation. The first group of containers are typically "node- based", meaning elements of a container are allocated in separate nodes that are linked together, while the second group of containers are "array-based", meaning elements of a container are allocated in contiguous storage. It's a lot easier to provide commit-or-rollback semantics on node-based containers than array-based ones; hence the two levels of guarantees.
So, while the above example for vectors is guaranteed to execute to completion, there's no way of knowing what the output will be. (The "0543211" output comes from the Silicon Graphics free STL, and looks to be the result of a partial resizing or copying operation; another STL implementation might produce an entirely different result).
There are some special cases to the general description of the two levels of guarantees above. For instance: multiple element insertion operations on maps and sets do not have the first level guarantee. Insertions of PODs - plain old C-level structs - for vectors and deques do have it. Stacks and queues have it as well. Thus for complete details you'll have to check the standard or a reference book.
In conclusion, the basic benefit of all this is that if you have classes that use exception handling, you can put them into standard library containers and get reasonable and useful behavior in the event an exception is thrown.

auto_ptr

Jonathan Schilling, jls@sco.com
This newsletter has not yet mentioned the auto_ptr class found in the new standard library. This template class provides a simple form of local, exception-safe, dynamic memory allocation. Its design has undergone some changes during the standardization process, but is now final.
To understand the purpose of auto_ptr, first consider this code:

```
class SomeClass { ... void foo(); ... };
```

```
void f() {
    SomeClass* p = new SomeClass();
    p->foo();
}
```

Calling function f() causes a memory leak, because the storage acquired by the new operator is never released and the destructor for SomeClass (which may itself release other acquired storage or resources within SomeClass) is never called.
Even if the coding of f() is changed to:

```
void f() {
    SomeClass* p = new SomeClass();
    p->foo();
    delete p;
}
```

the function may still leak, because if an exception is thrown out of the call to foo(), the delete statement will never execute.
There are several approaches to getting around this type of problem but the one that best preserves the structure of the code is to use the standard library's auto_ptr class, like this:

```
#include <memory>
using namespace std;

void f() {
    auto_ptr<SomeClass> p(new SomeClass());
    p->foo();
}
```

auto_ptr is a template class that is instantiated with the class being pointed to as its argument type. Objects of the auto_ptr class are initialized by a regular pointer to the class being pointed to. Once created, auto_ptr objects are used just like a regular pointer: that is, the * and -> operations are defined for auto_ptr objects, with practically no additional overhead over their built-in versions.
But most importantly, it is not necessary to write a delete statement to correspond to the new operator above; rather, as part of the destruction of the auto_ptr object at the end of its scope (function f(), in this case), the memory acquired by the new will be deleted, and the SomeClass destructor will be called. And since destructors are called for local objects when exceptions are thrown and the stack is unwound (see C++ Newsletter #017), the same will happen if an exception is thrown by the call to foo().
In addition, the auto_ptr class has a member function get() which allows you to get at the original regular pointer, and member functions release() and reset() which allow you to explicitly deassociate or delete the original pointer. These are less often used, however.
The design changes in auto_ptr have come from deciding whether to allow auto_ptr objects to be copy constructed or assigned. In particular, the former is necessary if auto_ptr objects are to be passed to or returned from functions. From the first public Committee Draft to the second Draft to the final Draft International Standard, this part of auto_ptr has changed, and so what may be in the standard library implementation you are currently using may not yet correspond to the final version of the library adopted by the committee at the Morristown meeting this past November.

In the final version, copy construction and assignment of auto_ptr objects is allowed, but only on non-const objects. Copying an auto_ptr transfers "ownership" of the storage pointed to by the auto_ptr to the destination, and the source of the copy is modified so that its pointer is null. Thus, in the following case:

```
void g(auto_ptr<SomeClass> p) {
    p->foo();    // ok
}

void f() {
    auto_ptr<SomeClass> p(new SomeClass());
    g(p);
    p->foo();    // runtime error
}
```

the call to foo() in g() will work properly, but the one in f() would cause runtime undefined behavior (such as a core dump) because it would be a dereference of a null pointer. These "destructive copy semantics" are of course different from normal pointer copying semantics (where both the destination and source point to the same storage) and are the reason why only non-const objects may be used.
Because of these transfer of ownership semantics, it is generally not possible to use auto_ptr objects within collections, such as STL. This is because the algorithms that implement STL may create temporary copies (for example, during a sort) that unexpectedly gain ownership of the auto_ptr, causing very unpredictable results in the container.
You may have seen "smart pointer" classes elsewhere which present alternative interfaces, semantics, and implementations. It's important to realize that the standard library auto_ptr class is NOT a generalized, all-purpose smart pointer class, nor is it a substitute for garbage collection. Rather, it is designed for one specific purpose, and the "auto" part of the auto_ptr name should be considered suggestive: use auto_ptr for local (automatic) variables and temporaries only.

C++ and Signal Handling

Jonathan Schilling, jls@sco.com
In C++ Newsletter issue #015 it was mentioned that C++ exceptions are not related to operating system signals. In other words, if an operating system signal occurs (such as an attempt to reference an invalid address, or an interrupt keystroke from the terminal running the program), it will not be mapped into a C++ exception; rather, the signal will have to be handled by the techniques described in the C standard library, which is included by reference into the C++ standard library. These C standard library techniques are defined by the <signal.h> header (which becomes header <csignal> in the C++ standard library).
A call to the signal() function defines a signal handler function to be given control in the event a particular signal is raised. The C standard constrains the behavior of this handler function in

a number of respects -- it can only terminate in certain ways, it usually can't call other standard library functions, it can't refer to static objects unless they are of type volatile sig_atomic_t, and so forth (see Section 7.7.1.1 of the ISO C standard for full details). Any violation of these constraints results in undefined behavior.

This left open the question of what constraints a C++ signal handling function has. The standards committee resolved this during the past year by adding language to the standard defining the concept of a "plain old function" (POF), analogous to the existing concept of plain old data (POD). A POF is a function that only uses the common subset of the C and C++ languages. A C++ signal handler only has defined behavior if it is a POF and if it would have defined behavior under the C standard; in particular, any handler which is not a POF -- i.e., which uses any C++ features -- will have undefined behavior.

This means that even innocuous uses of C++ language features, such as "for (int i ...)" instead of "for (i ...)", in a signal handler will result in undefined behavior, even when in implementation terms it should make no difference.

That's the letter of the law. In practice, things are different.

First, the C standard is not really the arbiter of signal handlers. Typically operating systems provide more full-featured, robust signal handling mechanisms than the bare-bones level of portable support given by the C standard. Similarly, the constraints upon signal handlers in operating systems may be different. For instance, the POSIX standard has a more specific and generally less restrictive set of constraints upon signal handlers than the C standard. It is this operating system definition or standard that then often becomes the important one for programmers to be aware of.

Second, a lot of C++ features do not impact the runtime considerations of what would interfere with signal handling. Examples would include interspersing of declarations with statements, use of references, use of scoping notation (assuming the scope reference itself was well-defined), and many others. Such innocuous usages, while strictly speaking undefined by the C++ standard, are very likely to work without problems.

The C++ features that are not likely to work are those that involve complicated runtime processing or state information. In particular, any use of C++ exception handling within a signal handler is likely to lead to disaster (and indeed a footnote in the standard points this out), unless the implementation documentation has specifically stated that it will work. Similarly, runtime type information (RTTI) and static object declaration with dynamic initialization are good candidates for malfunction.

Note the C++ standard does not say that use of C++ features in signal handlers is "implementation-defined", in which case implementations would have to document which features will or will not work in a signal handler, but rather "undefined", which lets C++ vendors off the hook. It is up to the user of an implementation to figure out whether a particular C++ feature can safely operate in a signal handler, hopefully with some general guidance from the vendor. Of course, the safest route is to follow the letter of the law and comply with the C standard restrictions by avoiding C++ features entirely.

The Vector Constructor Ambiguity Problem

Jonathan Schilling, jls@sco.com

The vector container of the C++ standard library was introduced in C++ Newsletter issue #015. From its inception in the original Standard Template Library specification, a declaration such as

```
vector<int> v(100, 1);
```

meant "construct a vector of 100 integers, all initialized to 1".
Then later, a member template constructor was added, so that you could construct a vector from a copy of another container, given the beginning and ending of an input iterator range. A typical usage would be:

```
list<int> li;
...
vector<int> v2(li.begin(), li.end());
```

However it turns out that adding the declaration for this member template overloads the original declaration. This can be seen from this example (simplified from the real vector class):

```
#include <stdio.h>

template <class T>
class vector {
public:
    typedef unsigned int size_type;
    vector(size_type, T) { printf("ctor 1\n"); }
    template<class II> vector(II first, II last)
                { printf("ctor 2\n"); }
};

int main() {
    vector<int> v(100, 1);      // expecting ctor1
    return 0;
}
```

This program prints out "ctor 2", because that is the better match (the first constructor requires a type conversion), but it is certainly not what the user intended! Other usages can lead to compile-time ambiguities, usually with fairly incomprehensible error messages.
Now one way to work around the problem in this instance would be to simply insert a cast to the actual type in the first constructor:

```
vector<int> v((vector<int>::size_type)100, 1);
```

With this change the above program prints out "ctor 1", since that is now the better match. But this is non-intuitive, and furthermore the ANSI/ISO standards committee discovered that this problem also occurs in other sequence containers in the standard library, such as list, deque, and basic_string, and in other member functions, such as insert() and replace(), whenever a template argument type convertible to int was involved. So a better resolution was needed.

The solution adopted by the committee last year was simply to declare that the second constructor shall have the effect of the first constructor, if the input iterator type (class II in the above example) is an integral type! In other words, the library will "do what I mean", not "do what I say".
How is this done in the library? One way is to specialize the member template for every integral type. But the standard also mysteriously states that "Less cumbersome implementation techniques also exist". This is referring to implementing a compile-time dispatching scheme inside the library, whereby the implementation can tell whether the instantiating type of the second constructor is integral or not. This involves clever uses of partial specialization and the numeric_limits<T> traits class. Bjarne Stroustrup stated during a committee meeting that people who look at the code for this technique react with "fascinated horror", but fortunately the horror is for standard library vendors and not you!
Meanwhile, if you do not have access to a standard library implementation that conforms to the final standard, you may have to use a work-around such as presented above.

Removal of Error-Prone Default Arguments

Jonathan Schilling, jls@sco.com
Say you're using a not-quite-up-to-date version of the C++ standard library and you see this code:

```
vector<int> v(22);
...
v.assign(7);
```

Without having to get out your nearest STL book, what would you think the second statement does?
A reasonable guess might be "assign 7 to every element of v". But that would be wrong. The old definition of this assign member function was essentially:

```
void vector<T>::assign(size_type n, const T& t = T()) {
    erase(begin(), end());
    insert(begin(), n, t);
}
```

So what the above code would really do is replace the existing vector with a vector of 7 elements, each of value zero. The zero comes from the default argument T(), which is int() in this case, which means a zero-initialized value.
This usage of a default argument in the library's specification was deemed "error-prone" by the ANSI/ISO C++ standards committee and was removed from the then-draft standard last year. So now, the assign member function is essentially;

```
void vector<T>::assign(size_type n, const T& t);
```

and a usage such as the above would have to be of the form:

```
v.assign(7, 0);        // assign vector of 7 elements, each zero
```

which is similar in appearance to the commonly-used constructor form:

```
vector<int> v2(7, x);   // construct vector of 7 elements, each x
```

so that the possibility of misunderstanding is much less likely.
The committee also removed similarly error-prone default arguments for the insert() member function and in the deque, list, and string classes.
In part this review of the standard was motivated by problems related to whether default argument expressions are instantiated if not used (see C++ Newsletters #024 and #025). But this issue also illustrates a more general design point that is pertinent to your programming as well: be judicious with default arguments, and take care to see whether function calls that omit the default arguments might get misinterpreted by human readers.

Typename Changes

Jonathan Schilling, jls@sco.com
This newsletter has not previously mentioned the "typename" keyword. This language feature was introduced several years ago during the standardization process. To understand its purpose, consider the following code:

```
template<class T> class Y {
        T::A a;             // error
};
```

When the compiler sees this class template definition, it has no way of knowing what T::A represents. In particular, it doesn't know whether T::A is a type or is something else. Usages such as

```
        T::A(bb);
```

might either be a function call of T::A passing global variable bb as an argument, or a declaration of a variable bb of type T::A. (Yes, in C and C++ you can declare variables that way; makes parsing lots of fun!)
Issue #017 of the Newsletter discussed the idea of dependent and non-dependent names within templates. T::A is a dependent name (because part of the name is the template formal parameter T), and the language rule became that a dependent name within a template is assumed to NOT be a type unless the applicable name lookup finds a type (which it doesn't here) or unless the new typename keyword is used. Neither of these happens in this case.
So, the above examples need to be modified to

```
template<class T> class Y {
        typename T::A a;       // ok
        typename T::A(bb);     // ok, bb is a data member
    };
```

and all is well.
Since typename was introduced, its permitted use has been expanded a couple of times to make template writing easier. A while ago, the language was changed to allow typename to appear before any qualified name, even if the name isn't template dependent, as long as the usage is within the scope of a template declaration or definition.

More recently, at the last committee meeting before the standard was made final, the language syntax was revised to allow typename to be used within a return statement. This makes the following usage possible:

```
class V {
public:
	typedef int weight;
	// ...
};

class W {
public:
	typedef int weight;
	// ...
};

template <class T>
class A {
public:
	typename T::weight f() {
		return typename T::weight();	// now allowed
	}
	// ...
};

void z() {
	A<V> a;
	a.f();
}
```

The first use of typename corresponds to what we've discussed above. But without also having typename in the return statement, the compiler would have to assume that that T::weight (being a dependent name) was a function name rather than a type, and so would generate an error.

As a temporary source of confusion, some compilers have not yet implemented all of the consequences of dependent/non-dependent lookups in templates, and so this example might compile even without the typename. Also note that as a non-standard extension some compilers will assume an implicit typename in the example at the beginning, causing it to compile without a typename.

More importantly, the above example is the sort of architecture you see in STL or generic programming, where several classes share the same characteristic (in this case, a type named "weight"), and you can make use of that characteristic without knowing which of those classes you're being instantiated with. It is in such circumstances that the typename keyword is most likely to be necessary.

Object-oriented Design

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 1 - ABSTRACTION

Up until now we've largely avoided discussing object-oriented design (OOD). This is a topic with a variety of methods put forward, and people tend to have strong views about it. But there are some useful general principles that can be stated, and we will present some of them in a series of articles.

The first point is perhaps the hardest one for newcomers to OOD to grasp. People will ask "How can I decide what classes my program should have in it?" The fundamental rule is that a class should represent some abstraction. For example, a Date class might represent calendar dates, an Integer class might deal with integers, and a Matrix class would represent mathematical matrices. So you need to ask "What kinds of entities does my application manipulate?"

Some examples of potential classes in different application areas would include:

GUI/Graphics - Line, Circle, Window, TextArea, Button, Point

Statistics - Mean, ChiSquare, Correlation

Geography - River, Country, Sea, Continent

Another way of saying it would be this. Instead of viewing an application as something that performs steps A, B, and C, that is, looking at the program in terms of its functions, instead ask what types of objects and data the application manipulates. Instead of taking a function-oriented approach, take an object-oriented one.

One obvious question with identifying potential classes is what level of granularity to apply. For example, in C++ an "int" is a primitive type, that represents an abstraction of mathematical integers. Should int be a class in the usual C++ sense? Probably not, because a class implies certain kinds of overhead in speed and space and in user comprehension. It's interesting to note that Java(tm), a newer object-oriented language, also has int, but additionally supports a "wrapper" class called Integer that represents an integer value. In this way, an application can manipulate integers either as primitives or as classes.

Consider a slightly more ambiguous case. Suppose that you're writing a Date class, and you want to express the concept "day of week". Should this be a class of its own? Besides devising a class for this purpose, at least five other representations are possible:

```
int dow : 3;  (bit field)
```

```
char dow;
```

```
short dow;
```

```
int dow;
```

```
enum Dow {SUN, MON, TUE, WED, THU, FRI, SAT};
```

The "right" choice in this case is probably the enumeration. It's a natural way of representing a limited domain of values

Direct use of primitive types for representation has its drawbacks. For example, if I choose to represent day of week as an integer, then what is meant by:

```
int dow;

...

dow = 19;
```

The domain of the type is violated. As another example, C/C++ pointers are notorious for being misused and thereby introducing bugs into programs. A better choice in many cases is a higher-level abstraction like a string class, found in the C++ and Java standard libraries.
On the other end of the scale, it's also possible to have a class try to do too much, or to cover several disparate abstractions. For example, in statistics, it doesn't make sense to mix Mean and Correlation. These statistical methods have little in common. If you have a class "Statistics" with both of these in it, along with an add() member function to add new values, the result will be a mishmash. For example, for Mean, you need a stream of single values, whereas for Correlation, you need a sequence of (X,Y) pairs.
We will have more to say about OOD principles. A good book illustrating several object-oriented design principles is "Designing and Coding Reusable C++" by Martin Carroll and Margaret Ellis, published by Addison-Wesley.

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 2 - DATA ABSTRACTION

As we said in the previous issue, object-oriented design has many aspects to it, and a variety of strong views about which approach is "right". But there are some general techniques that are useful.
One of these, one that constitutes a whole design method in itself, is data abstraction. Simply stated, data abstraction refers to identifying key data types in an application, along with operations that are to be done on those types.
What does this mean in practice? Suppose that we are doing graphics of some sort, and are concerned with X,Y points on a screen. Now, at a low enough level, a point might be described via a couple of floating-point numbers X and Y. But with data abstraction, we define a type "Point" that will refer to a point, and we hide from the users of the type just how such a point is implemented. Instead of directly using X,Y values, we present Point as a distinct data type, along with some operations on it.
In the case of a Point type, two of those operations are (1) establishing a new Point instance, that describes an actual screen point, and (2) computing the distance between this point and another point.
If Point was written out as a C++ class, it might look like:

```
class Point {
    float x;
    float y;
public:
    Point(float, float);
    float dist(const Point&);
};
```

We've declared a class Point with a couple of private data members. There is a constructor to create new object instances of Point, and a member function dist() to compute the distance between this point and another one.
Suppose that we instead implemented this as C code. We might have:

```
struct Point {
    float x;
    float y;
};

typedef struct Point Point;

float Point_dist(Point*);
```

and so on.
The C approach will certainly work, so why all the fuss about data abstraction and C++? There are several reasons for the fuss. One is simply that data abstraction is a useful way of looking at the organization of a software program. Rather than decomposing a program in terms of its functional structure, we instead ask the question "What data types are we operating on, and what sorts of operations do we wish to do on them?"
With data abstraction, there is a distinction made between the representation of a type, and public operations on and behavior of that type. For example, I as a user of Point don't have to know or care that internally, a point is represented by a couple of floating-point numbers. Other choices might conceivably be doubles or longs or shorts. All I care about is the public behavior of the type.
In a similar vein, data abstraction allows for the formal manipulation of types in a mathematical sense. For example, suppose that we are dealing with screen points in the range 0-1000, typical of windowing systems today. And we are using the C approach, and say:

```
Point p;

p.x = 125;
p.y = -59;
```

What does this mean? The domain of the type has been violated, by introduction of an invalid value for Y. This sort of invalid value can easily be screened out in a C++ constructor for Point. Without maintaining integrity of a type, it's hard to reason about the behavior of the type, for example, whether dist() really does compute the distance appropriately.
Also, if the representation of a type is hidden, it can be changed at a later time without affecting the users of the type.
As another simple example of data abstraction, consider designing a String class. In C, strings are implemented simply as character pointers, that is, of type "char*". Such pointers tend to be error prone, and we might desire a higher-level alternative.
In terms of the actual string representation, we obviously have to store the string's characters, and we also might want to store the string length separately from the actual characters.
Some of the operations on strings that we might want would include:

- creating a String from a char*

- creating a String from another String

- retrieving a character at a given index

- retrieving the length

- searching for a pattern in a String

Given this very rough idea for a data type, we could write C++ code like so:

```
class String {
    char* str;
    int len;
public:
    String(const char*);
    String(const String&);
    char charAt(int) const;
    int length() const;
    int search(const String&) const;
};
```

and so on.

In medium-complexity applications, data abstraction can be used as a design technique by itself, building up a set of abstract types that can be used to structure a complete program. It can also be used as part of other design techniques. For example, in some application I might have a calendar date type, used to store the birthdate of a person in a personnel record. Data abstraction could be used to devise a Date type, independent of any other design techniques used in the application.

There is an excellent (but out of print) book on data abstraction, with the title "Abstraction and Specification in Program Development", by Barbara Liskov and John Guttag (published 1986 by MIT Press). Note also that data abstraction is only one part of object-oriented design and programming. Some languages (Modula-2, Ada 83) support data abstraction without being fully object-oriented.

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 3 - POLYMORPHISM

The example in the previous section illustrates another aspect of object-oriented design, that of polymorphism. This term means "many forms", and in the context that we are using refers to the ability to call member functions of many object types using the same interface.

The simplest C++ example of this would be:

```
#include <iostream.h>

class A {
public:
    virtual void f() {cout << "A::f" << endl;}
};

class B : public A {
public:
    virtual void f() {cout << "B::f" << endl;}
};

int main()
```

```
{
    B b;
    A* ap = &b;

    ap->f();

    return 0;
}
```

which calls B::f(). That is, the base class pointer ap "really" points at a B object, and so B::f() is called.

This feature requires some run-time assistance to determine which type of object is really being manipulated, and which f() to call. One implementation approach uses a hidden pointer in each object instance, that points at a table of function pointers (a virtual table or vtbl), and dispatches accordingly.

Without language support for polymorphism, one would have to say something like:

```
#include <iostream.h>

class A {
public:
    int type;
    A() {type = 1;}
    void f() {cout << "A::f" << endl;}
};

class B : public A {
public:
    B() {type = 2;}
    void f() {cout << "B::f" << endl;}
};

int main()
{
    B b;
    A* ap = &b;

    if (ap->type == 1)
        ap->f();
    else
        ((B*)ap)->f();

    return 0;
}
```

that is, use an explicit type field. This is cumbersome.

The use of base/derived classes (superclasses and subclasses) in combination with polymorphic functions goes by the technical name of "object-oriented programming".

It's interesting to note that in Java, methods (functions) are by default polymorphic, and one has to specifically disable this feature by use of the "final", "private", or "static" keywords. In C++ the default goes the other way.

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 4 - DATA HIDING

Another quite basic principle of object-oriented design is to avoid exposing the private state of an object to the world. Earlier we talked about data abstraction, where a user-defined type is composed of data and operations on that data. For example, in C++ a type Date might represent a user-defined type for calendar dates, and operations would include comparing dates for equality, computing the number of days between two dates, and so on.
Suppose that in C++, a Date type looks like this:

```
class Date {
public:
    int m;      // month 1-12
    int d;      // day 1-31
    int y;      // year 1800-1999
};
```

and I say:

```
Date dt;

dt.m = 27;
```

What does this mean? Probably nothing good. So it would be better to rewrite this as:

```
class Date {
    int m;
    int d;
    int y;
public:
    Date(int, int, int);
};
```

with a public constructor that will properly initialize a Date object.
In C++, data members of a class may be private (the default), protected (available to derived classes), or public (available to everyone).
A simple and useful technique for controlling access to the private state of an object is to define some member functions for setting and getting values:

```
class A {
    int x;
public:
    void set_x(int i) {x = i;}
    int get_x() {return x;}
};
```

These functions are inline and have little or no performance overhead.
In C++ there is another sort of hiding available, that offered by namespaces. Suppose that you have a program with some global data in it:

```
int x[100];
```

and you use a C++ class library that also uses global data:

```
double x = 12.34;
```

These names will clash when you attempt to link the program. A simple solution is to use namespaces:

```
namespace Company1 {
    int x[100];
}

namespace Company2 {
    double x = 12.34;
}
```

and refer to the values as "Company1::x" and "Company2::x". Note that the Java language has no global variables, and similar usage to this example would involve static data defined in classes.

Data hiding is a simple but extremely important concept. Without it, it is difficult to reason about the behavior of an object, given that its state can be arbitrarily changed at any point.

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 5 - REPRESENTATION HIDING

In the last issue we talked about data hiding, where the internal state of an object is hidden from the user. We said that one reason for this hiding is so that the state can not be arbitrarily changed.

Another aspect of hiding concerns the representation of an object. For example, consider a class to handle a stack of integers:

```
class Stack {
    ???
public:
    void push(int);
    int pop();
    int top_of_stack();
};
```

It's pretty obvious what the public member functions should look like, but what about the representation? At least three representations could make sense. One would be a fixed-length array of ints, with an error given on overflow. Another would be a dynamic int array, that is grown as needed by means of new/delete. Yet a third approach would be to use a linked list of stack records. Each of these has advantages and disadvantages.

Suppose that the representation was exposed:

```
class Stack {
public:
    int vec[10];
    int sp;
    ...
    int top_of_stack();
};
```

and someone cheats by accessing top of stack as:

```
obj.vec[obj.sp]
```

instead of:

```
obj.top_of_stack()
```

This will work, until such time as the internal representation is changed to something else. At that point, this usage will be invalidated, and will not compile or will introduce subtle problems into a running program (what if I change the stack origin by 1?).
The point is simply that exposing the internal representation introduces a set of problems with program reliability and maintainability.

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 6 - EXTENSIBILITY

Thus far we've looked at object-oriented design in isolation, focusing on individual classes as abstractions of some real-world entity. But as you're probably already aware, C++ classes (and ones in other languages as well) can be extended by deriving subclasses from them. These classes add functionality to the base class.
Suppose that we have a class:

```
class A {
private:
    int x;
protected:
    int y;
public:
    int z;
};
```

The declarations of the members indicate that x is available only to member functions of the class itself, y is available to subclasses, and z is available to everyone.
How do we decide how to structure a class for extensibility? There are several aspects of this, one of them being the level of protection of individual members. There is not a single "right" answer to this question, but one approach is to ask how the class is likely to be used. For example, with a Date class:

```
class Date {
private:
    long repr;          // days since 1/1/1800
};
```

it's unlikely that a derived class will need to directly access repr, because it's in an arcane format and because the Date class can supply a set of functions that will suffice to manipulate Dates. There is a steep learning curve in learning how to directly manipulate the underlying representation, and a consequent ability to mess things up by getting it wrong.
On the other hand, for a Tree class:

```
class TreeNode {
protected:
    TreeNode* left;
    TreeNode* right;
    int value;
public:
    TreeNode(TreeNode*, TreeNode*, int);
};
```

making the internal pointers visible may make sense, to facilitate a derived class walking through the tree in an efficient manner.

It's useful to distinguish between developers, who may wish to extend a class, and end users. For example, with the Date class, the representation (number of days since 1/1/1800) is non-standard, and in a hard format to manipulate. So it makes sense to hide the representation completely. On the other hand, for TreeNode, with binary trees as a well-understood entity, giving a developer access to the representation may be a good idea.
There's quite a bit more to say about extensibility, which we will do in future issues.

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 7 - MORE ABOUT EXTENSIBILITY

In the last issue we started talking about extending the functionality of a base class via a derived class. Another aspect of this simply deals with the issue of how far to carry derivation. In other words, how many levels of derived classes make sense?
In theoretical terms, the answer is "an infinite number". That is, you can carefully design a set of classes, with each derived class adding some functionality to its base class. There is no obvious stopping point for this process.
In practical terms, however, deep class derivations create more problems than they solve. At some point, humans lose the ability to keep track of all the relationships. And there are some hidden performance issues, for example with constructors and destructors. The "empty" constructor for C in this example:

```
#include <iostream.h>

class A {
public:
	A() {cout << "A::A\n";}
	~A() {cout << "A::~A\n";}
};

class B : public A {
public:
	B() {cout << "B::B\n";}
	~B() {cout << "B::~B\n";}
};

class C : public B {
public:
	C() {cout << "C::C\n";}
	~C() {cout << "C::~C\n";}
};

void f()
{
	C c;
}

int main()
{
```

```
        f();

        return 0;
    }
```

in fact causes the constructors for B and A to be called, and likewise for the destructor.
As a simple rule of thumb, I personally try to keep derivations to three levels or less. In other words, a base class, and a couple of levels of derived classes from it.

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 8 - A BOOK ON C++ DESIGN

In issue #027 the new edition of Bjarne Stroustrup's book "The C++ Programming Language" was mentioned. This book came out a few months ago, and contains about 100 pages on design using C++. It starts out by discussing design at an abstract level, and then moves on to cover specific design topics as they relate to C++. Recommended if you're interested in this topic.

INTRODUCTION TO OBJECT-ORIENTED DESIGN PART 9 - TEMPLATES VS. CLASSES

In previous issues we've looked at some of the aspects of template programming. One big issue that comes up with object-oriented design is when to implement polymorphism via a template (a parameterized class or function) in preference to using inheritance or a single class.
This is a hard question to answer, but there are several aspects of the issue that can be mentioned. Consider first the nature of the algorithm that is to be implemented. How generally applicable is the algorithm? For example, sorting is used everywhere, and a well-designed sort function template for fundamental or user-defined types would be very handy to have around. On the other hand, consider strings. Strings of characters are very heavily used in programming languages. But what about strings of doubles? For example, does taking a substring of doubles from a string mean very much? In certain applications it might, but clearly this feature is not as generally useful as strings of characters.
On the other side of this same argument, if we want to implement an algorithm for a set of types, and some of those types are much more widely used than others (such as a string of chars), then template specializations offer a way to tune performance. For example, a string template might be defined via:

```
    template <class T> class string { ... };
```

with a specialization for characters:

```
    class string<char> { ... };
```

that is optimized. Of course, if strings of chars represent 99% of the use of the string template, then perhaps simply devising a string class would make more sense.
Another question to ask is whether all the types of interest fit cleanly into a single class hierarchy. For example, a hierarchy for a GUI window system might have:

```
    class Component { ... };

    class Container : public Component { ... };
```

```
class Window : public Container { ... };

class Frame : public Window { ... };
```

That is, all types are in one hierarchy. Such a type hierarchy is often best managed via abstract classes and virtual functions, without the use of templates. Note that using virtual functions allows for access to runtime type information, whereas templates are more of a compile-time feature. Newsletter issues #024, #025, and #026 give some examples of the use of virtual functions and runtime type identification.

But sometimes templates might be useful even in a simple hierarchy such as this one. For example, a hierarchy of GUI classes might be parameterized based on the type of the underlying display device, such as a bit-mapped display, dumb terminal, or touch-screen.