

New-Style C++: a Beginning

NewStyle C++ 1

Old- versus New-Style Header Files
Header File Content Gotchas
New Implementation Gotchas
Why??
How??
Namespaces and "using"
Namespaces and Standard Headers
What does this buy you?

Old- versus New-Style Header Files

NewStyle C++ 2

The adoption of the C++ Standard officially brought a number of changes to the language. One of the most visible was the creation of a new set of header files (largely motivated by the adoption of the namespace mechanism).

A C++ programmer is now confronted with two sets of standard header files, related by a naming convention, and with a host of similarities and differences:

Old style:	New style:
<code>iostream.h</code>	<code>iostream</code>
<code>fstream.h</code>	<code>fstream</code>
<code>string.h</code>	<code>string</code>
<code>math.h</code>	<code>cmath</code>
<code>stdlib.h</code>	<code>cstdlib</code>

In general, old-style C++ header files are replaced by new-style headers whose names omit the ".h" suffix. Some headers, such as `math.h`, were inherited from the C language. In those cases, the new-style headers prefix a "c" to the name and omit the ".h".

Header File Content Gotchas

NewStyle C++ 3

In general, the corresponding old- and new-style header files declare more-or-less the same types and serve the same purpose. However, there are a number of important exceptions. A sampling:

<code>iostream.h</code> standard stream stuff	<code>iostream</code> same type names, but some subtle differences in implementation
<code>fstream.h</code> file stream stuff; includes <code>iostream.h</code>	<code>fstream.h</code> file stream stuff; does NOT include <code>iostream.h</code>
<code>string.h</code> C-style char arrays	<code>string</code> string object library

Do not make the mistake of assuming that this is a complete list of the issues.

New Implementation Gotchas

NewStyle C++ 4

There are also implementation differences (improvements, **really!**) that can cause problems. For instance:

```
// Example: ostream gotcha
#include <fstream>
using namespace std;

void printHeader(ofstream Out);

void main() {
    ofstream oFile("dump.txt");
    printHeader(oFile);
    oFile.close();
}

void printHeader(ofstream Out) {
    Out << "header" << endl;
}
```

This program will cause a runtime exception, after writing to the output file and after closing the stream.

The reason is a deadly combination:

- Out is passed by value
- no deep copy constructor is provided for the `ofstream` class

The error does not occur when using the old-style header files. Evidently, this is because there is a deficiency in the old-style copy constructor for `ofstream` objects.

Moral: ALWAYS pass stream objects by reference.

Why??

NewStyle C++ 5

Who are we to question the 9-year deliberations of the C++ Standards Committee?
Seriously, it doesn't matter. The fact is that we have to deal with the situation as it is.

A few observations:

- The new-style headers offer enhanced functionality.
- There are some S/E advantages incorporated into the new-style implementation.
- Therefore, use the new-style approach whenever possible.
- Never, ever, mix old- and new-style headers in the same compilation unit. If possible don't mix them in the same program.

How??

NewStyle C++ 6

Two differences are immediately obvious:

```
#include <fstream>
using namespace std;

void main( ) {
    int anInt;
    ifstream inStream;
    ofstream outStream;
    inStream.open("infile.dat");
    outStream.open("outfile.dat");

    inStream >> anInt;
    while (inStream) {
        outStream << anInt << endl;
        inStream >> anInt;
    }

    inStream.close( );
    outStream.close( );
}
```

header file name omits ".h"

"using directive" makes it possible to refer to the identifiers declared in the header file.

Namespaces and "using"

NewStyle C++ 7

A namespace is a scope with a name attached. That is:

```
namespace FooSpace {
    typedef struct {
        string Message;
        int Target;
    } Foo;
    const int MaxFoo = 1000;
    int numFoo;
    Foo List[MaxFoo];
};
```

A namespace may contain declarations and/or definitions. The elements of a namespace can only be accessed by using one of several syntactic structures:

```
. . .
cout << FooSpace::numFoo;
. . .
```

```
using FooSpace::numFoo;
cout << numFoo;
cout << List[0].Message;
```

Error. List[] is not declared in the present scope.

```
using namespace FooSpace;
cout << numFoo;
cout << List[0].Message;
```

Namespaces and Standard Headers

NewStyle C++ 8

The new-style C++ header files are all wrapped in a single namespace, called `std`:

```
// foobar
#ifndef FOOBAR
#define FOOBAR
namespace std {
    // declarations
}
#endif
```

Namespaces may be composed; that is, two with the same name are automatically concatenated by the preprocessor.

So it's not enough to `#include` the right header files; you also must make appropriate use of "using". For now, just apply a using directive as shown before.

What about the old-style headers?

They didn't escape:

```
// foobar.h
#ifndef FOOBAR_H
#define FOOBAR_H
namespace std {
    // declarations
}
using namespace std;
#endif
```

What does this buy you?

NewStyle C++ 9

Probably not much just yet. However, Stroustrup suggests the following approach:

Ideally, every entity in a program belongs to some recognizable logical unit ("module"). Therefore, every declaration in a nontrivial program should ideally be in some namespace named to indicate its logical role in the program. The exception is `main()`, which must be global in order for the run-time environment to recognize it as special.

In fact, global scope is itself considered a namespace, with no name (!). An undisciplined programmer can refer to a global identifier by prefixing the scope-resolution operator (`::`) to it even if there's a local declaration of the same name:

```
int Stupid = 0;
void F( ) {
    int Stupid = 10;
    cout << Stupid;    // local
    cout << ::Stupid; // global
}
```

Hint: you could wrap all those tempting globals into a namespace to protect them.