

Old- versus New-Style Header Files

Header File Gotchas

Why??

How??

Namespaces and “using”

Namespaces and Standard Headers

What does this buy you?

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:

`iostream.h`

`fstream.h`

`string.h`

`math.h`

`stdlib.h`

New style:

`iostream`

`fstream`

`string`

`cmath`

`cstdlib`

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”.

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:

`iostream.h`

standard stream stuff

`iostream`

same type names, but some subtle differences in implementation

`fstream.h`

file stream stuff;
includes `iostream.h`

`fstream.h`

file stream stuff; does NOT include `iostream.h`

`string.h`

C-style char arrays

`string`

string object library

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

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.

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.

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;  
. . .
```

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

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

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

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
```

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.