C/C++ Program Design

LAB 9

CONTENTS

- Learn to create multiple files
- Learn the concept of storage duration, scope and linkage
- Learn to use namespaces

2 Knowledge Points

2.1 Multiple-File Structure

2.2 Storage duration, Scope and Linkage

2.3 Namespaces

2.1 Multiple-File Structure

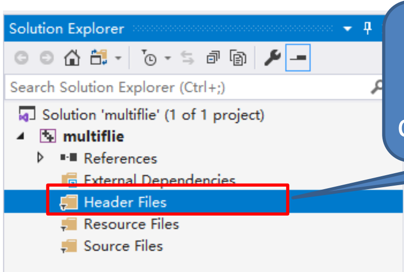
C++, like C, allows and even encourages you to locate the component functions of a program in separate files. You can compile the files separately and then link them into the final executable program. Using **make**, if you modify just one file, you can recompile just that one file and then link it to the previously compiled versions of the other files. This facility makes it easier to manage large programs.

You can divide the original program into **three parts**:

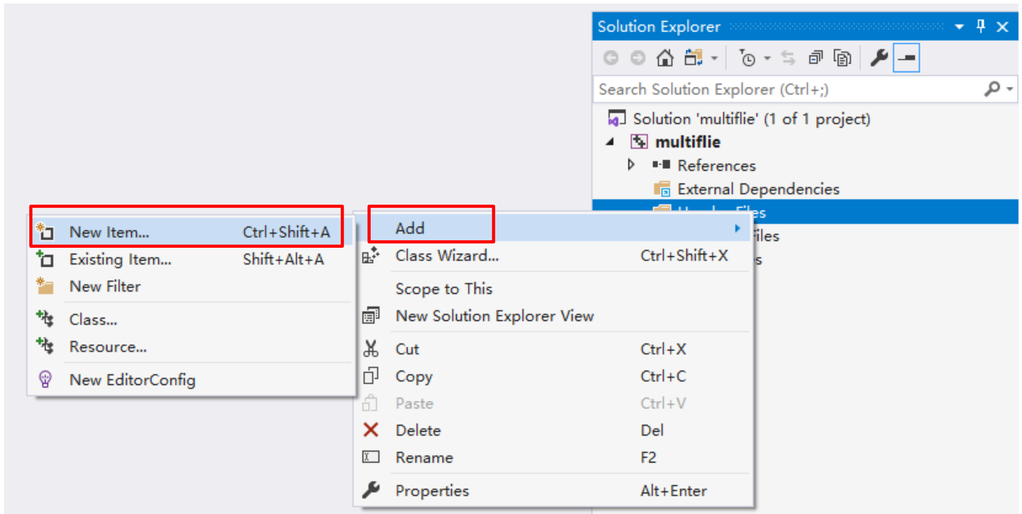
- **A header file** that contains the structure declarations and prototypes for functions that use those structures
- **A source code file** that contains the code for the structure-related functions
- **A source code file** that contains the code that calls the structure-related functions

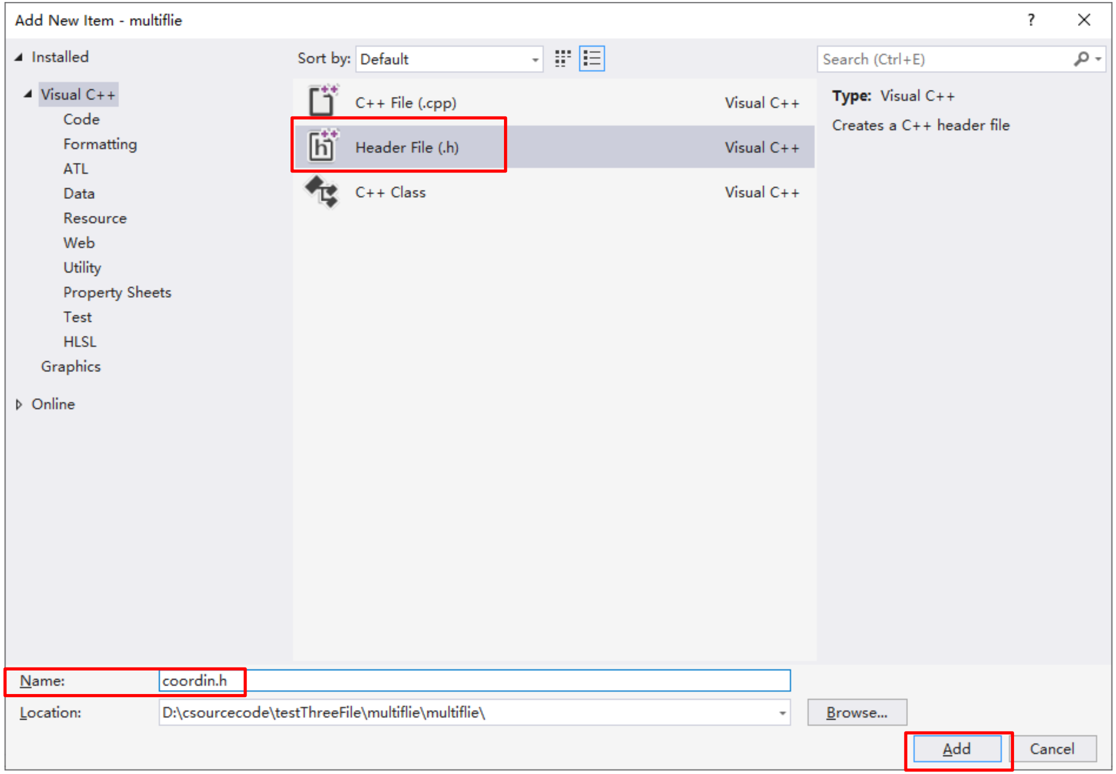
Example: three-file program

Create a new project in Visual Studio

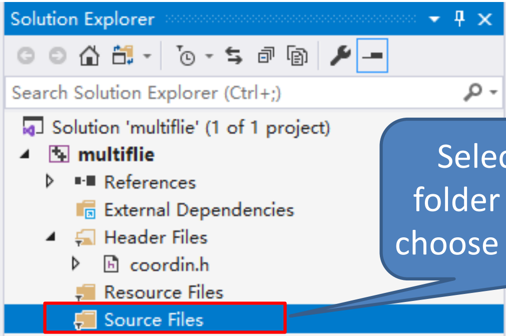


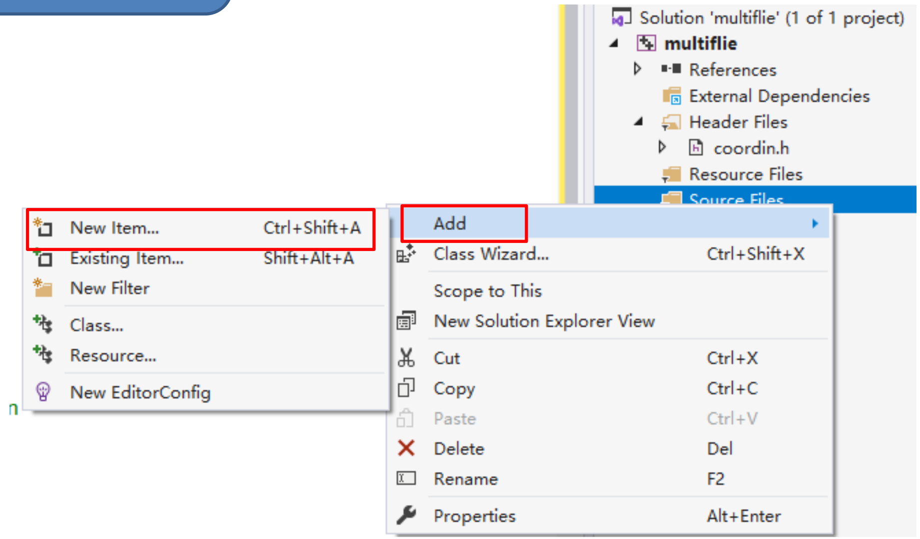
Select “Header Files” folder and right click and choose “add” →”New item”

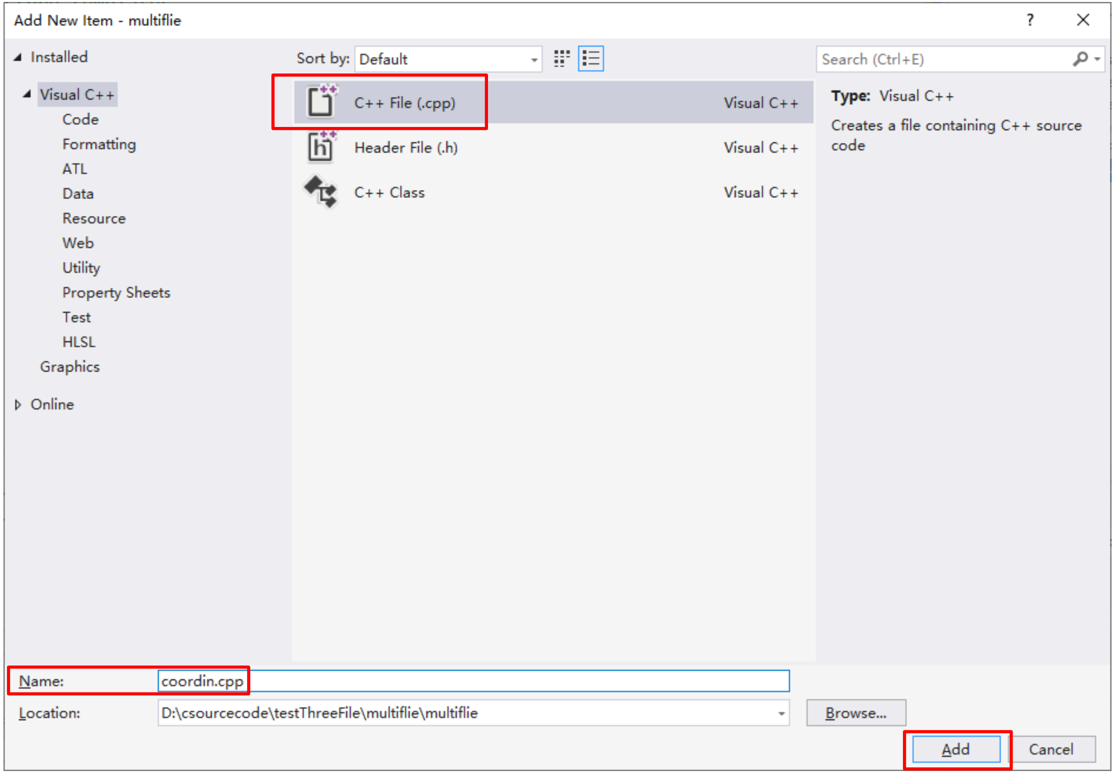



Add New Item - multiflie
Installed
Visual C++
Code
Formatting
ATL
Data
Resource
Web
Utility
Property Sheets
Test
HLSL
Graphics
Online
Sort by: Default
Search (Ctrl+E)
C++ File (.cpp)
Visual C++
Header File (.h)
Visual C++
C++ Class
Visual C++
Type: Visual C++
Creates a C++ header file
Name:
coordin.h
Location:
D:\csourcecode\testThreeFile\multiflie\multiflie\
Browse...
Add
Cancel

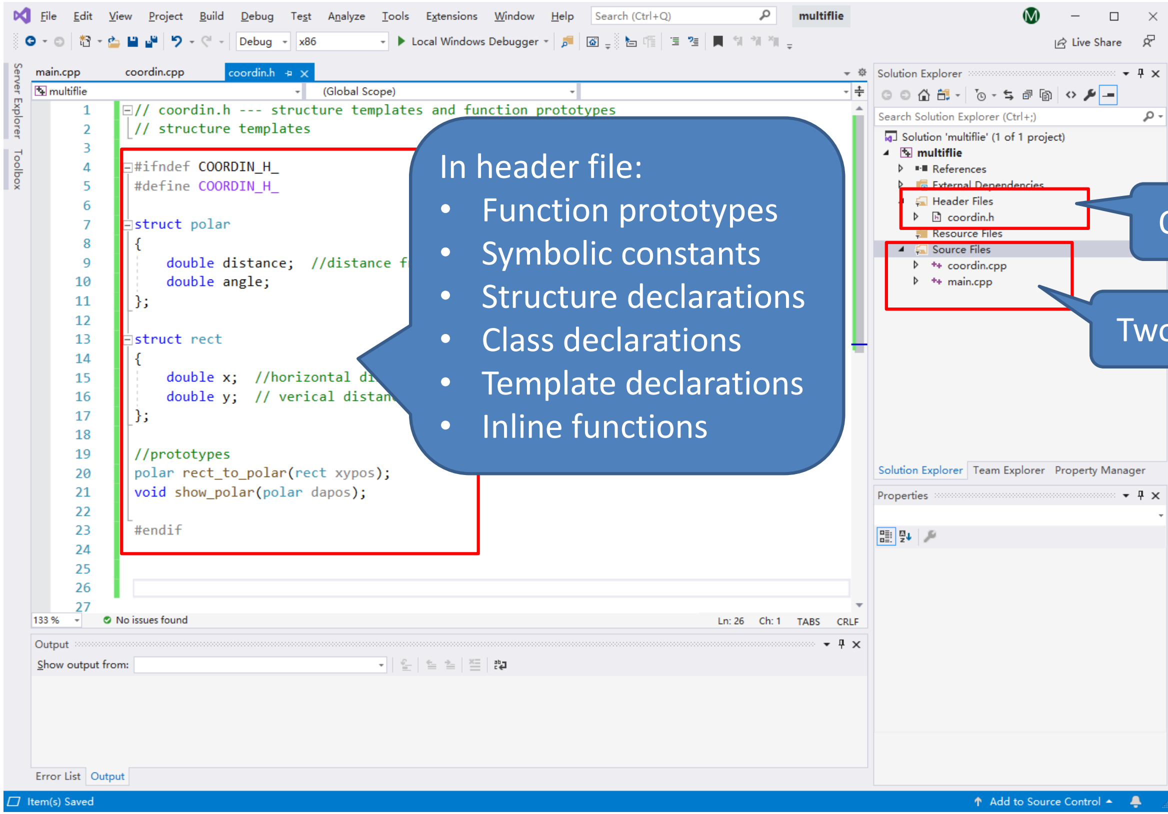
Select "Sources Files" folder and right click and choose "add" →"New item"






Add New Item - multiflie
Installed
Visual C++
Code
Formatting
ATL
Data
Resource
Web
Utility
Property Sheets
Test
HLSL
Graphics
Online
Sort by: Default
Search (Ctrl+E)
C++ File (.cpp)
Visual C++
Header File (.h)
Visual C++
C++ Class
Visual C++
Type: Visual C++
Creates a file containing C++ source code
Name:
coordin.cpp
Location:
D:\csourcecode\testThreeFile\multiflie\multiflie
Browse...
Add
Cancel

Example: three-file program



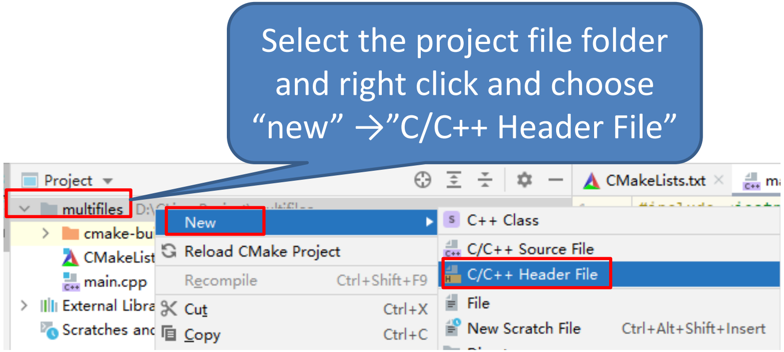
In header file:

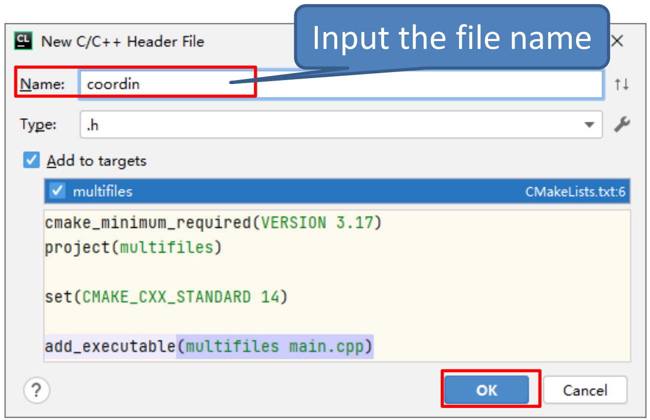
- Function prototypes
- Symbolic constants
- Structure declarations
- Class declarations
- Template declarations
- Inline functions

One header file

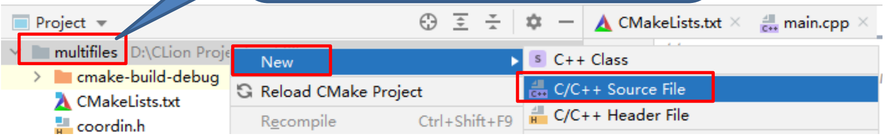
Two source files

Create a new project in CLion

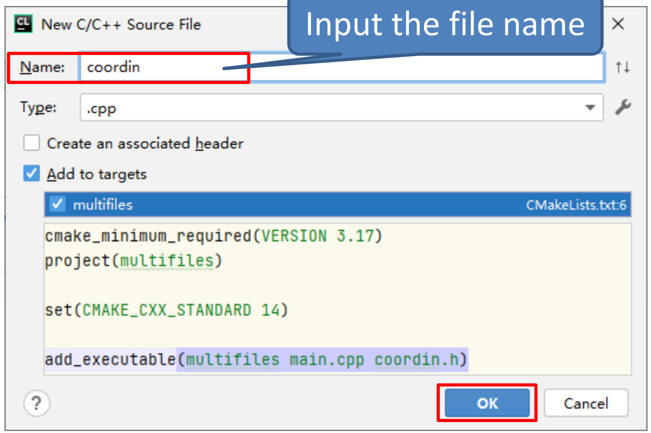


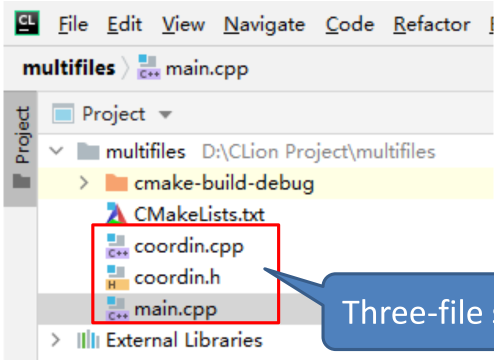


Select the project file folder and right click and choose “new” →”C/C++ Source File”



Input the file name





Three-file structure

CMakeLists.txt main.cpp coordin.h coordin.cpp

```
cmake_minimum_required(VERSION 3.17)
project(multifiles)

set(CMAKE_CXX_STANDARD 14)

add_executable(multifiles main.cpp coordin.h coordin.cpp)
```

CMakeLists.txt is created automatically

```
#ifndef COORDIN_H_
#define COORDIN_H_
// place include file contents here
#endif
```

Using preprocessor **#ifndef** directive to avoid multiple inclusions of header files.

You can also use preprocessor #pragma once to the same purpose.

In source file, you must include the header file.

```
#include <iostream>
```

Angle brackets for the standard header files

```
#include "coordin.h"
```

Double quotation marks for your header files

They have different search path. < > looks at t the part of the host system’s file system
“ “ looks at the current working directory and standard location

2.2 Storage duration, Scope and Linkage

C++ uses three separate schemes(four under C++11) for storing data.

- Automatic storage duration: **Variables declared inside a function definition**(including function parameters)have automatic storage duration. They are created when program execution enters the function or block in which they are defined, and the memory used for them is freed when execution leaves the function or block.

- Static storage duration: **Variables defined outside a function definition** or else by using the **keyword static** have static storage duration. They persist for the entire time a program is running.

- Dynamic storage duration: Memory allocated by the **new operator** persists until it is freed with the delete operator or until the program ends, whichever comes first. This memory has dynamic storage duration and sometimes is termed the free store or the heap.

- Thread storage duration(C++11): Variables declared with the **thread_local** keyword have storage that persists for as long as the containing thread lasts.

2.2.1 Automatic Storage Duration

Function parameters and variables declared inside a function have, by default, automatic storage duration. **They also have local scope and no linkage**.

```
int main()
{
    int teledeli = 5;
    {
        int websight = -2;    //websight scope begins
        cout << websight << endl;

        int teledeli = 0;     // the new definition hides the prior one
        cout << teledeli << endl;

    }                         //websight expires

    cout << teledeli << endl;

}
```

2.2.2 Static Storage Duration

C++, like C, provides **static storage duration variables** with three kinds of linkage: **external linkage** (accessible across files), **internal linkage** (accessible to functions within a single file),and **no linkage** (accessible to just one function or to one block within a function).

All three last for the duration of the program. The static variables stay present as long as the program executes.

1.Static Duration, External Linkage

Variables with **external linkage** are often simply called **external variables(global variables)**. They necessarily have static storage duration and file scope. External variables are defined outside, and hence external to, any function.

If you use an external variable in several files, only one file can contain a definition for that variable (per the one definition rule). But every other file using the variable needs to declare that variable using the keyword **extern**.

```
// file01.cpp
extern int cats = 20; // definition because of initialization
int dogs = 22;        // also a definition
int fleas;            // also a definition
...

// file98.cpp
// use cats, dogs, and fleas from file01.cpp
extern int cats;
extern int dogs;
extern int fleas;
...
```

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

int x;
int  main()
{
    int x = 256;

    cout << "local variable x = " << x << endl;

    cout << "global variable x = " << ::x << endl;

    return 0;
}
```

Declare a global variable whose initial value is 0

Declare a local variable whose name is the same as the global variable.

The local variable hides the global variable.

Using **scop-solution operator(::)** to access the global variable.

```
local variable x = 256
global variable x = 0
```

2. Static Duration, Internal Linkage

Applying the **static** modifier to a file-scope variable gives it internal linkage. A variable with internal linkage is local to the file that contains it. But a regular external variable has external linkage, meaning that it can be used in different files.

```
// file1
int errors = 20;        // external declaration
...
---------------------------------------------
// file2
int errors = 5;         // ??known to file2 only??
void froobish()
{
      cout << errors;   // fails
      ...
```

external variable

```
// file1
int errors = 20;        // external declaration
...
---------------------------------------------
// file2
static int errors = 5;  // known to file2 only
void froobish()
{
      cout << errors;   // uses errors defined in file2
      ...
```

Using **static** to share data among functions found in just one file, avoiding name conflicting with external variable.

3. Static Duration, No Linkage

You create such a variable by applying the **static** modifier to a variable defined **inside a block**. When you use it inside a block, static causes a local variable to have static storage duration. If you initialize a static local variable, the program **initializes the variable once**.

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

void Inc(void);
int  main()
{
    Inc();
    Inc();
    Inc();

    return 0;
}
void Inc(void)
{
    int x = 0;

    x++;

    cout << "x = " << x << endl;
}
```

auto variable

```
x = 1
x = 1
x = 1
```

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

void Inc(void);
int  main()
{
    Inc();
    Inc();
    Inc();

    return 0;
}
void Inc(void)
{
    static int x = 0;

    x++;

    cout << "x = " << x << endl;
}
```

Static variable

```
x = 1
x = 2
x = 3
```

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

long Factorial(int n);
int  main()
{
    int i;
    for (i = 1; i <= 5; i++)
        cout << i << "! = " << Factorial(i) << endl;

    return 0;
}
long Factorial(int n)
{
    static long product = 1;

    product *= n;

    return(product);
}
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

2.3 Namespace

The C++ Standard provides **namespace** facilities to provide greater control over the scope of names and avoid name conflicting.

This two variables are not conflict.

```
namespace Jack {
    double pail;                    // variable declaration
    void fetch();                   // function prototype
    int pal;                        // variable declaration
    struct Well { ... };            // structure declaration
}
namespace Jill {

    double bucket(double n) { ... } // function definition
    double fetch;                   // variable declaration
    int pal;                        // variable declaration
    struct Hill { ... };            // structure declaration
}
```

you can use ::, the scope-resolution operator, to qualify a name with its namespace.

```
Jack::pail = 12.34;  // use a variable
Jill::Hill mole;     // create a type Hill structure
Jack::fetch();       // use a function
```

using Declarations and using Directives

The **using declaration** lets you make particular identifiers available, and the **using directive** makes the entire namespace accessible.

```
namespace Jill {
    double bucket(double n)
    double fetch;
    struct Hill { ... };
}
char fetch;
int main()
{
    using Jill::fetch;    // put fetch into local namespace
    double fetch;         // Error! Already have a local fetch
    cin >> fetch;         // read a value into Jill::fetch
    cin >> ::fetch;       // read a value into global fetch
    ...
}
```

variable declared in namespace Jill

global variable

Using declaration

Placing a using declaration at the external level adds the name to the global namespace:

```
void other();
namespace Jill {
    double bucket(double n) { ... }
    double fetch;
    struct Hill { ... };

}
using Jill::fetch;   // put fetch into global namespace
int main()
{
    cin >> fetch;    // read a value into Jill::fetch
    other()
...
}

void other()
{
    cout << fetch;   // display Jill::fetch
...
}
```

A **using declaration**, makes a single name available. In contrast, the **using directive** makes all the names available.

```
using namespace Jack;  // make all the names in Jack available
```

```
#include <iostream>    // places names in namespace std
using namespace std;   // make names available globally

int main()
{
    using namespace jack; // make names available in main()
...
}
```

Generally speaking, the using declaration is safer to use than a using directive because it shows exactly what names you are making available.

```cpp
namespace sdm {
    const double BOOK_VERSION = 2.0;
    class Handle{...};
    Handle& getHandle();
}

void f1()
{
    using namespace sdm;

    cout << BOOK_VERSION;      // OK
    Handle h = getHandle();    // OK
}

void f2()
{
    using sdm::BOOK_VERSION;

    cout << BOOK_VERSION;      // OK
    Handle h = getHandle();    // wrong
}

void f3()
{
    cout << sdm::BOOK_VERSION;   // OK

    double d = BOOK_VERSION;     // wrong
    Handle h = getHandle();      // wrong
}
```