CSCI251: Advanced Programming

Lecturer: Dr. Shixun Huang

Exception and Namespaces

Spring 2024

Outline

- Exceptions:
  - Throwing and catching.
- Namespaces:
  - Scope.
  - Nested.
    - Inline.
  - Aliases.
- Programming defensively.
  - Briefly.

Traditional Error Handling

```
void inputStudentRec(Student &sRec) {
    int id, phone, day, month, year;
    string addr, name, email;
     …
    cout << "Date of birth (day month year):";
    cin >> day >> month >> year;
    if(day < 1 || day > 31)
        exit(1);
    if(month < 1 || month > 12)
        exit(1);
    …
}
```

Program ends abruptly ☹

- The **`exit()` function** forces the program to end.
  - Use a zero (0) argument (or return `EXIT_SUCCESS`) to indicate the program exited normally.
  - A non-zero argument (or return `EXIT_FAILURE`) is used to indicate an error has occurred in the program.

- The use of `exit` in functions is somewhat inflexible.
  - Invalid entries will result in a message and program termination.
- A function should be able to determine an error situation.
- Many programmers avoid such a sudden exit to the program.
  - It doesn't follow the concept of structured programming.
  - It may be hard to determine the cause.

- A better alternative (often):
  - Let a function detect an error.
  - Notify the calling function of the error.
  - Let the calling function determine what to do.

```
bool inputStudentRec(Student &sRec)
{
    bool errorCode = true;
    .  .  .
    if(day < 1 || day > 31)
        errorCode = false;
    .  .  .
    return( errorCode );
}
```

Throwing Exceptions

- Errors that occur during the execution of object-oriented programs are called **exceptions**.
  - They should be unusual occurrences.

- **Exception handling:**
  - This is an object-oriented technique to manage such errors
  - Throw an exception object.

- The actions you take with exceptions involve trying, throwing, and catching them:
  - You **`try`** a function; if it **`throws`** an exception, you **`catch`** and handle it.

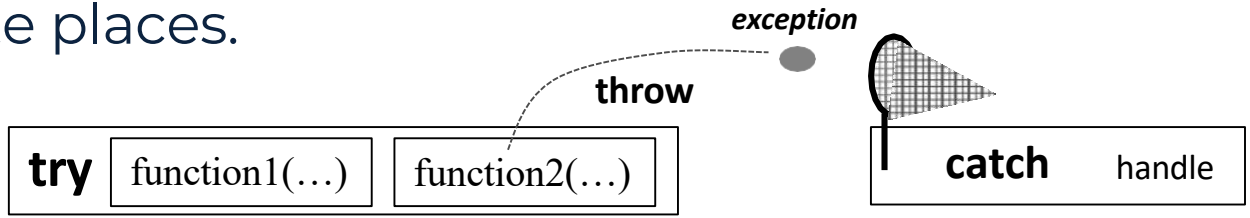
C++ exception keywords

- example:

```
try {
 //exceptions may be thrown using throw
}
catch( ){
        // handle your exception
    }
```

- If something may pose a problem
  - we should use the *try* block.
  - example

- We can **represent** an exception as an **object**.
- We **throw** an exception where an error occurs.
- We then **Catch & handle** the exception at appropriate places.



- **Exception:**
  - An *object*
  - It can be of any type, including a basic or class type.
  - A variety of exception object types can be sent from a function, regardless of its return type.

- Note that: a function should check for errors, but not necessary to handle an error.
- We separate error detection from error handling.
- Example: **throw an exception**.

```
void inputStudentRec() {
      int id, phone, day, month, year;
      string addr, name, email;
       …
      cout << "Date of birth (day month year):";
      cin >> day >> month >> year;
      if(day < 1 || day > 31)
           throw( string("Invalid day") );
      if(month < 1 || month > 12)
           throw( string( "Invalid month") );
      …
}
```

Catching Exceptions

- In the catch handler
  - free resources
  - do some cleaning up.
- **If an exception is not caught in your own program, the system will catch it and the default behaviour is to terminate the program.**

Catching and handling exceptions

- In the calling function use try-catch block to catch and handle exceptions.

```
int main() {
    Student stu1;

    try {
        inputStudentRec(stu1);
    }
    catch(string err) {
        cout << "error: " << err << endl;
    }
    stu1.display();
}
```

We **must** include curly braces, **even if** only one statement is tried

A **`try` block** consists of one or more function calls which the program attempts to execute, but which might result in thrown exceptions.

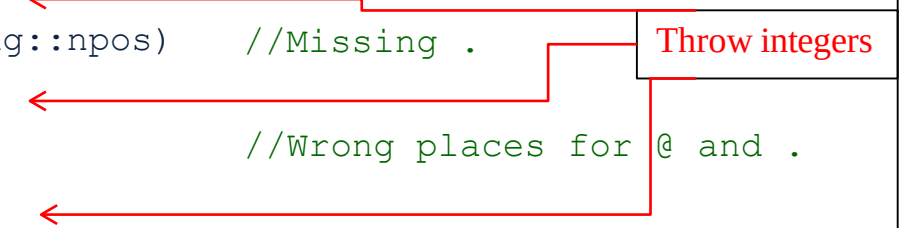
The exception handlers defined in the catch blocks.

Throwing multiple exceptions

- One function can throw multiple exceptions.

```
// Validate email address, email should be of the form x@y.z
void verifyEmail(string email) {
    unsigned int loc1, loc2;
    string at = "@";
    string dot = ".";

    loc1 = email.find(at);
    loc2 = email.rfind(dot);
    if(loc1 == string::npos)    //Missing @
        throw(1);
    if(loc2 == string::npos)    //Missing .
        throw(2);
    if(loc1 >= loc2)            //Wrong places for @ and .
        throw(3);
}
```



- One function can throw multiple types of exceptions.

```
void inputStudentRec(Student &sRec) {
    int id, phone, day, month, year;
    string addr, name, email;
    …
    cout << "Date of birth (day month year):";
    cin >> day >> month >> year;
    if(day < 1 || day > 31)
        throw(string("Invalid day"));
    if(month < 1 || month > 12)
        throw(month);
    cout << "email:";
    cin >> email;
…
}
```

Throw a string

Throw an integer

Catching multiple exceptions

```
int main() {
    Student stu1;

    try {
        inputStudentRec(stu1);
    } catch( string err ) {
        cout << "error: " << err << endl;
    } catch( int eno ) {
        if(eno == 1)
            cout << "error 1: No @ in email" << endl;
        else if(eno == 2)
            cout << "error 2: Not . in email" << endl;
        else if(eno == 3)
            cout << "error 3: @ before ." << endl;
        else
            cout << "Something wrong." << endl;
    }
}
```

Catching string exceptions.

Catching integer exceptions.

Rethrowing an Exception

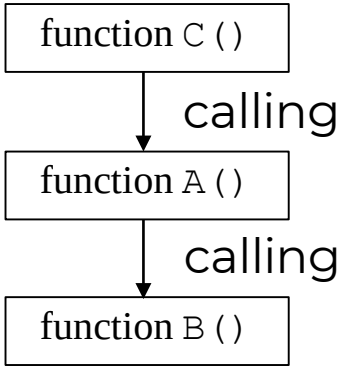
- It is possible the handler that catches an exception decides it cannot process the exception, or it may simply want to release resources before letting someone else handle it.

- In this case, the handler can simply rethrow the exception with the statement:

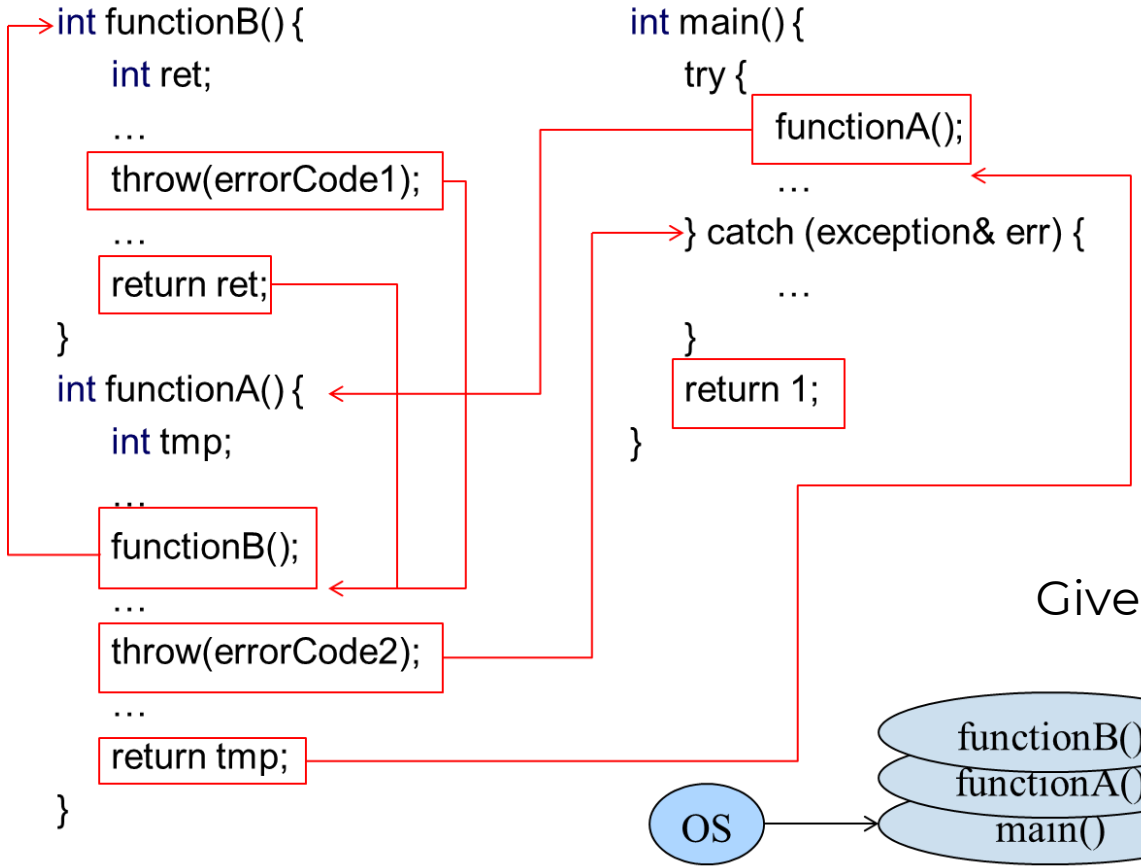
```
catch(…) {
    cout<<"An Exception was thrown"<<endl;
    // deallocate resource here, then rethrow
    throw;
}
```

Unwinding the Stack

- Your function A can `try` a function call B and, if the function B throws an exception, you can catch the exception.
- If your function A doesn't catch the exception, then a function C that calls your function A can still catch the exception.
  - If no function catches the exception, then the program terminates.
- This process is called **unwinding the stack**.



Unwinding the stack (continue)



Give a try!

Exceptions … they'll be back …

- While you can throw around any types, typically you make use of subclasses of the `exception` class that you have defined.
  - Once we've covered some basics on classes we will return to exceptions.

Namespaces

- These are optional scopes, accessed using the scope resolution operation `::` … as in `std::cout`.

```
using namespace std;
using std::cout;
```

- They help limit concerns about naming clashes, since we can distinguish between versions by referencing the scope/namespace they appear in.
  - We can use namespaces to manage different versions for example.

Syntax …

- To declare a name space of our own we would typically do the following:

```
namespace name-of-namespace {
    // declarations
}
```

- Using `using` brings a namespace into scope, or part of it anyway.
  - Once in scope, you can access something without needing the scope resolution operators.

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

namespace NS {
    int i;
}
// There is a gap here
namespace NS {
    int j;
}

int main ()
{
    NS::i=NS::j=10;
    cout << NS::i * NS::j << endl;

    using namespace NS;
    cout << i*j << endl;
    return 0;
}
```

This program produces the following output:

100

100

Scoping and namespaces …

counter.h

```
#ifndef _COUNTER_H_
#define _COUNTER_H_
int upperbound;
int lowerbound;
class counter {
    public:
          counter(int n) {
                if(n<=upperbound) count = n;
                else count = upperbound;
        }
        void reset(int n){
                if(n<=upperbound) count=n;
        }
        int run() {
                if(count > lowerbound) return count--
                ; else return lowerbound;
        }
    private:
        int count;
};
#endif
```

useCounter.cpp

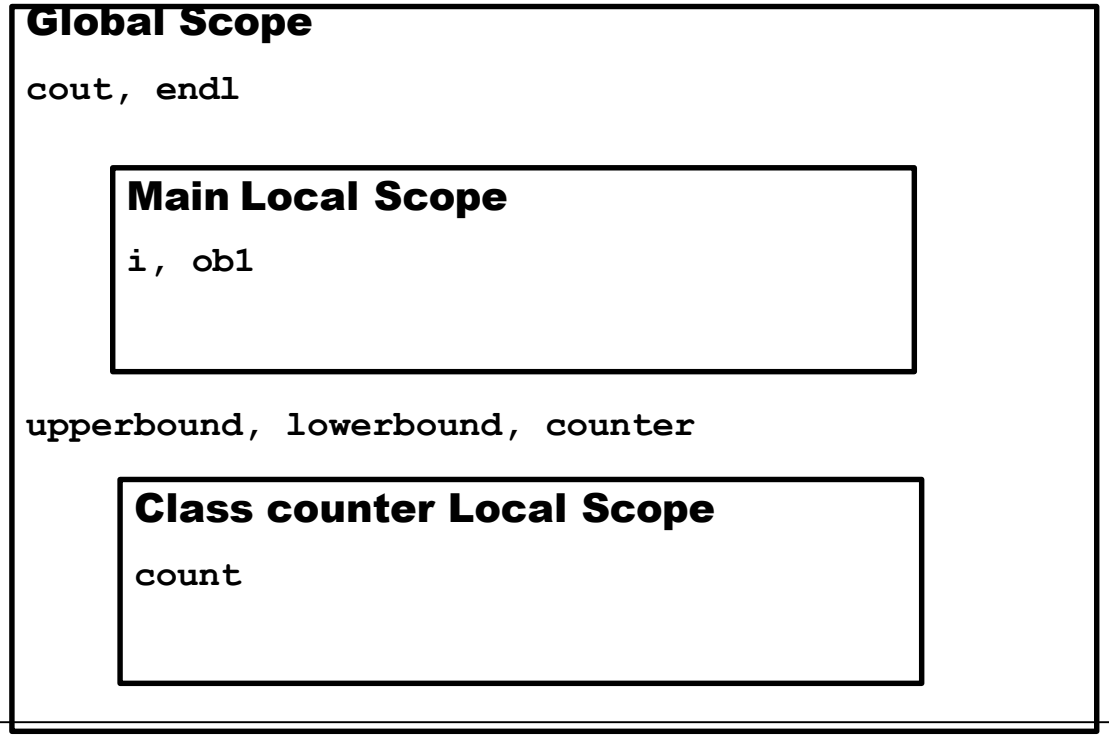
```
#include <iostream>
#include "counter.h"
using namespace std;

int main()
{
    upperbound = 5000;
    lowerbound = 1;
    counter ob1(10);

    int i=0;
    cout<<"Counter Object:";
    do {
        i = ob1.run();
        cout<<i<<" ";
    } while (i>lowerbound);
    cout<<endl;
}
```

This will use the constructor for the counter class.

- Scope of variables:
  - We know that variables can have global or local scope.



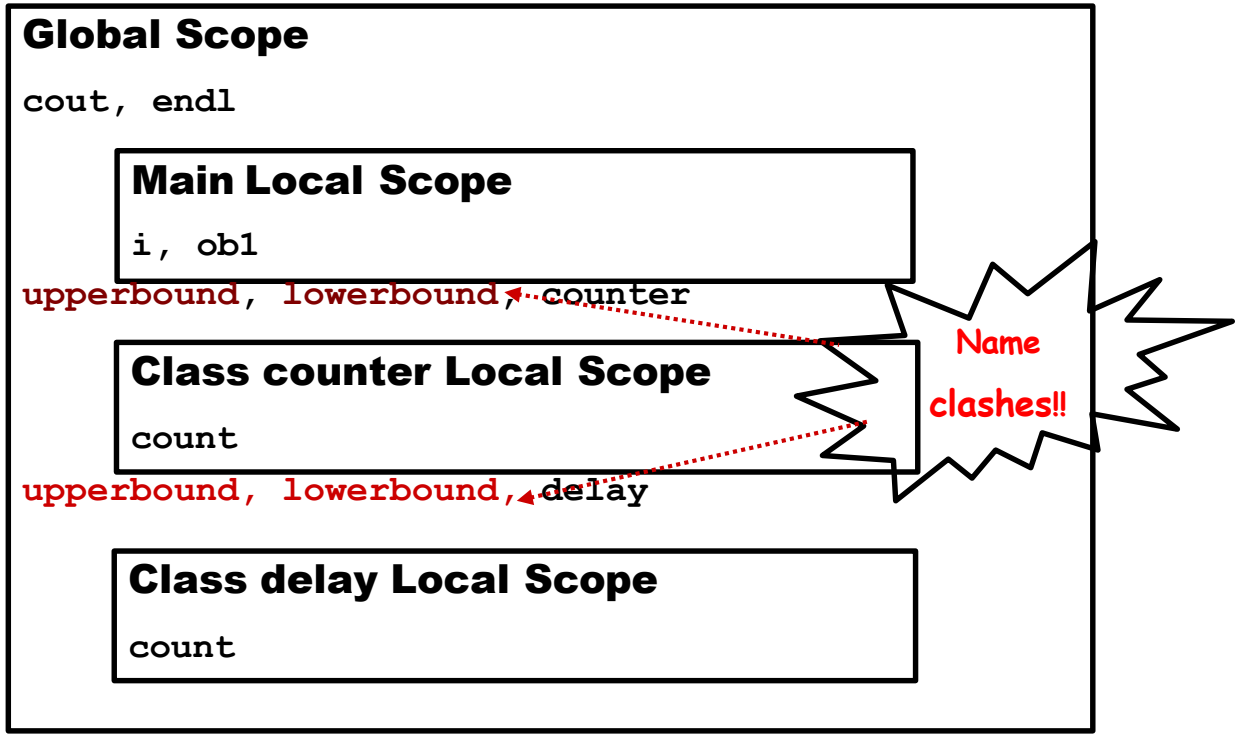
delay.h ---------->

```
#include <iostream>
using namespace std;
#include "counter.h"
#include "delay.h"
int main()
{
    ...;
}
```

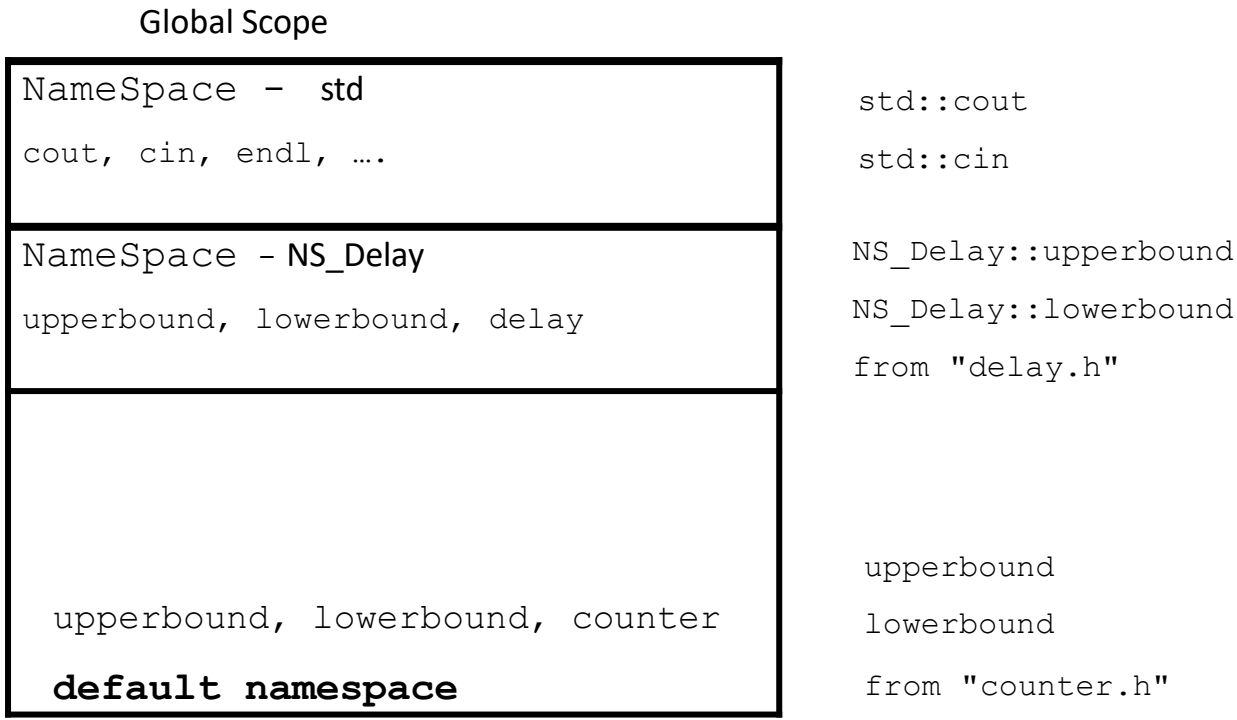
```
#ifndef _DELAY_H_
#define _DELAY_H_
int upperbound;
int lowerbound;
class delay {
   private:
        int count;

   public:
        delay(int n) {
                if(n<=upperbound) count =
                n; else count = upperbound;
        }
        void reset(int n){
                if(n<=upperbound) count=n;
        }
        int run() {
                if(count > lowerbound) return count--
                ; else return lowerbound;
        }
};
#endif
```

"delay.h", line 3: Error: Multiple declaration for upperbound.
"delay.h", line 4: Error: Multiple declaration for lowerbound.
2 Error(s) detected.


Global Scope
cout, endl
Main Local Scope
i, ob1
upperbound, lowerbound, counter
Class counter Local Scope
count
upperbound, lowerbound, delay
Class delay Local Scope
count
Name
clashes!!

- We avoid the clash as follows:



```
#ifndef _DELAY_H_
#define _DELAY_H_
namespace NS_Delay {
int upperbound;
int lowerbound;
class delay {
    public:
        delay(int n) {
                if(n<=upperbound) count =
                n; else count = upperbound;
        }
        void reset(int n){
                if(n<=upperbound) count=n;
        }
        int run() {
                if(count > lowerbound) return count--
                ; else return lowerbound;
        }
    private:
     int count;
};
}
#endif
```

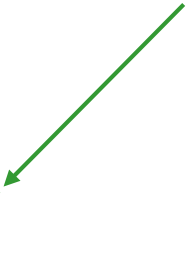
```
#include <iostream>
#include "counter.h"
#include "delay.h"

using namespace std;

int main()
{
    upperbound = 5000;
    lowerbound = 1;
    counter ob1(10);

    NS_Delay::upperbound = 100;
    NS_Delay::lowerbound = 1;
    NS_Delay::delay ob2(10);

    ...
```



defined in counter.h

defined in delay.h

namespace: NS_Delay

```
    int i = 0;
    cout<<"Counter Object:";
    do {
        i = ob1.run();
        cout << i << " ";
    } while ( i > lowerbound );
    cout << endl;


    cout<<"Delay Object:");
    do {
        i = ob2.run();
        cout << i << " ";
    } while ( i > NS_Delay::lowerbound );
    cout << endl;
}
```

Note

- Here `upperbound`, `lowerbound` and class `delay` are part of the scope defined by `NS_Delay` namespace.

- We can directly use them, without any namespace qualification:

```
if (count > lowerbound) return count--;
```

  - Those variables are within scope in the namespace.

- outside that namespace
  - to assign the value 10 to upperbound from code outside NS_Delay, you must use

    ```
    NS_Delay::upperbound = 10;
    ```
  - To declare an object of type `delay` from outside `NS_Delay`, you will use

    ```
    NS_Delay::delay ob;
    ```

Note

- Don’t put …

```
using namespace whatever;
```

- … in a header file because it will affect all code afterwards and cannot be undone.
  - See the next slide for localising…

- Don’t confuse namespaces with classes…

```
date::year …
```

- Is date a class or a namespace?

Using a namespace locally

```
#include<iostream>

void func1() {
    using namespace std;
    cout << "This is func1" << endl;
}

void func2() {
    std::cout << "This is func2" << std::endl;
}

int main() {
    func1();
    func2();

    std::cout << "This is Main" << std::endl;
    return 0;
}
```

Unnamed Namespaces

- There is a special type of namespace, called an `unnamed namespace`, also called `anonymous namespace`, They have this general form

```
namespace {
    // declarations
}
```

- Only working within the scope of a single file, i.e. within the file that contains the unnamed namespace.
  - This can provide a sort of encapsulation.
  - Members of that namespace can be used directly, without qualification.

Nested Namespaces

- A namespace must be declared outside of all other scopes.
  - This means you cannot declare namespaces that are localized to a function.
- However, a namespace can be nested within another.
- Namespace definitions hold declarations.
  - A namespace definition is a declaration itself, so namespace definitions can be nested.

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

namespace NS1 {
    int i;
    namespace NS2 { // a nested namespace
        int j;
    }
}

int main ()
{
    NS1::i=19; NS1::NS2::j=10;
    cout<<NS1::i * NS1::NS2::j<<endl;
    // use NS1 namespace
    using namespace NS1;
    // Now NS1 is in view, NS2 can be used to refer j
    cout<<i*NS2::j<<endl;
    return 0;
}
```

Inline namespaces:

- Names in an inline namespace can be used directly in the enclosing namespace.
- Example:

```
inline namespace Embedded{
…
}
```

Inline namespaces:

```
namespace Parent
{
    namespace Child1
    {       struct child1_data{int a;} ;
    }
    namespace Child2
    {       struct child2_data{int b;} ;
    }
    namespace child3
    {
        child1_data  data1;
        child2_data  data2;
    }
```

```
namespace Parent
{
     inline namespace Child1
    {       struct child1_data{int a;} ;
    }
    inline namespace Child2
    {       struct child2_data{int b;} ;
    }
    namespace child3
    {
        child1_data  data1;
        child2_data  data2;
    }
}
```

Namespace aliases

```
namespace University_of_Wollongong {
  int student();
}

namespace UOW =
  University_of_Wollongong;
```

- We are specifying an abbreviation we can use.

Aliasing for nested namespaces

- An alias can also be applied to a nested namespace.

```
namespace University_of_Wollongong {
   int student();
   namespace Nest_SCIT; {
    void a() { j++; }
    int j;
    void b() { j++; }
   }
}
namespace SCIT = University_of_Wollongong::Nest_SCIT;
```