# CSCI251/CSCI851 Autumn-2020
# Advanced Programming **(S3e)**

## Getting organised V:
## Exceptions (Part 1), namespaces and defensive programming

# Outline

- Exceptions (Part One):
  - 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 EXIT_SUCCESS) to indicate the program exited normally.
  - A non-zero argument (or 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, but not necessarily take action.
- 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 what caused the program to exit.

- 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, although it doesn’t just work with objects/classes, you can use built-in types.
- 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.

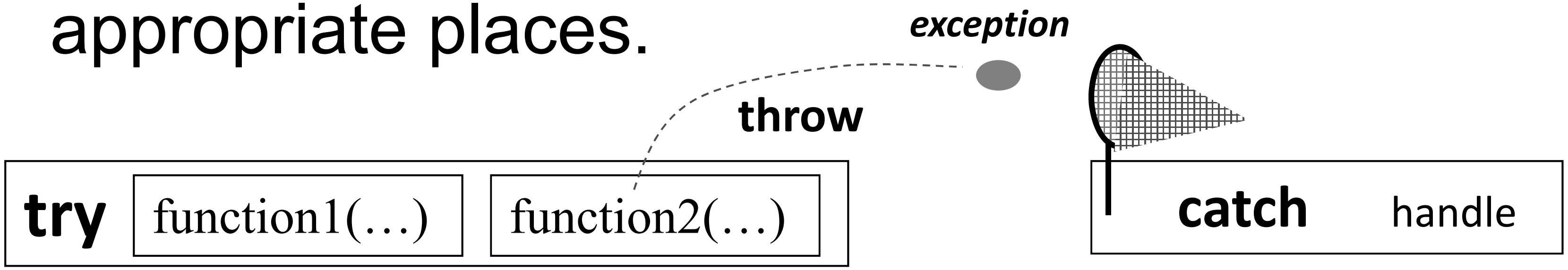
# C++ exception keywords

- When we use exceptions in C++ we must become familiar with the following keywords:

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

- If our program is potentially going to attempt doing something which may pose a problem we should embed the code in a *try* block.

- 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

throw

try function1(…) function2(…)

catch handle

- **Exception:** An *object* that contains information that is passed from the place where a problem occurs to another place that will handle the problem.
  - 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.

- A function should check for errors, but shouldn’t be required to handle an error if one is found.
- When a program detects an error within a function, the function should send an error object to the calling function, or **throw an exception**.

```
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( string( "Invalid month") );
      …
}
```

# Typically …

- … exceptions allow the person who writes a library function to detect errors and …
- … the user of the library function to decide how to handle those errors.
- We separate error detection from error handling.

# Catching Exceptions

- To handle a thrown object, you include one or more `catch` blocks immediately following a `try` block.
- In the catch handler you normally find code which will free resources or do some cleaning up.
  - Calling functions from within `catch` blocks can be dangerous, especially if you call the function that caused the thrown exception in the first place.
- **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;
    }
    stu2.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);
}
```

Throw integers

- 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(string("Invalid month"));
    cout << "email:";
    cin >> email;
    verifyEmail(email);
    …
}
```

Throw a string

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

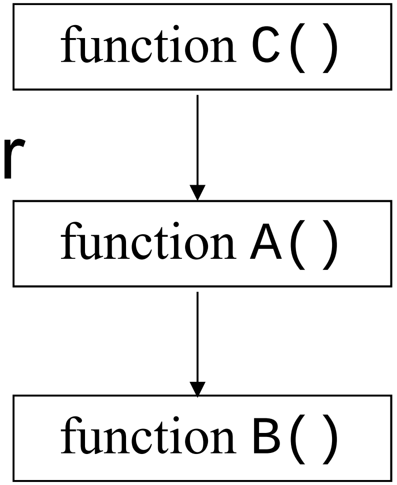
- It is possible the handler that catches an exception decides it cannot process the exception, or it may simply want to release resources before letting someone else handle it.

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

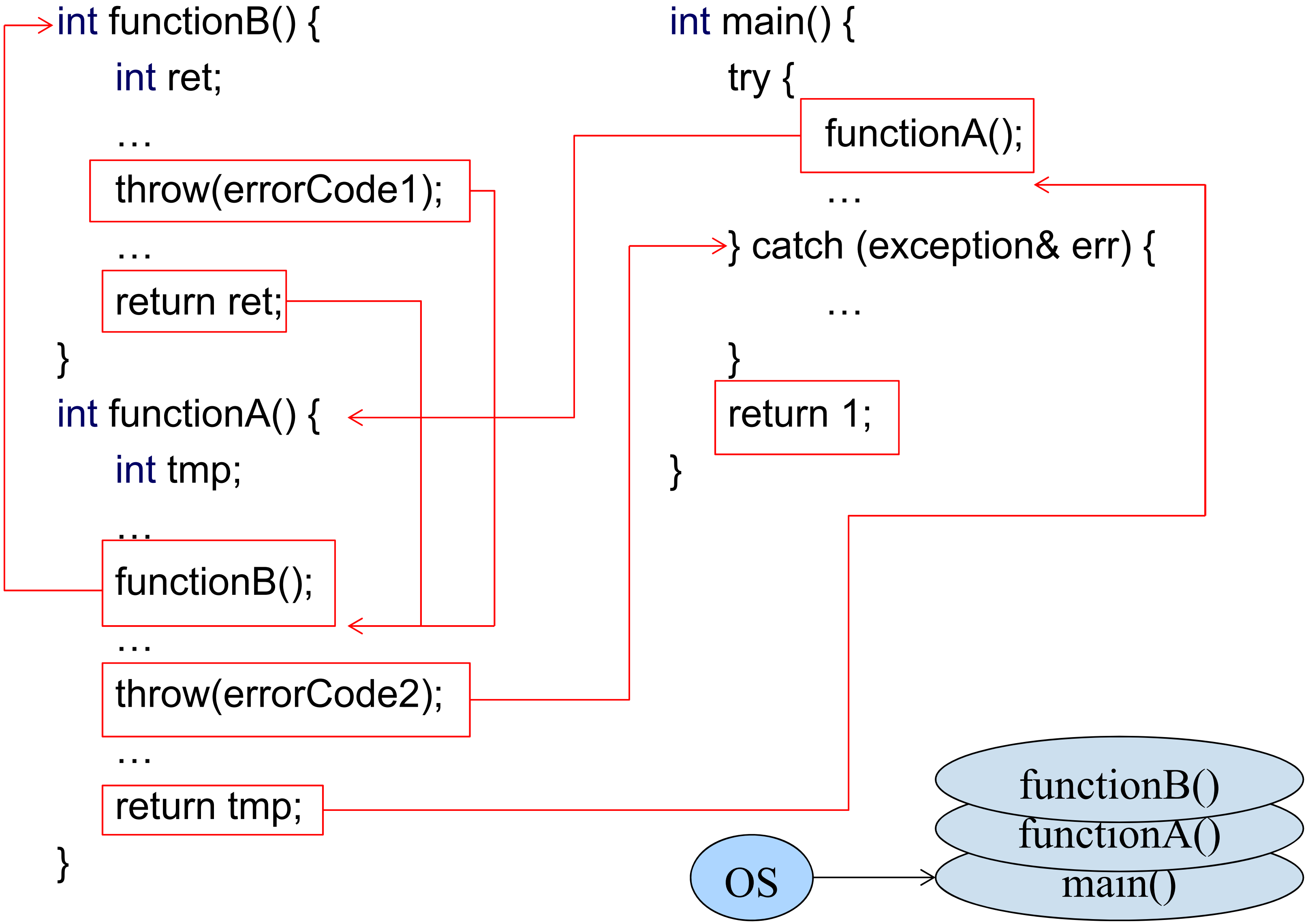
```
catch(…) {
   cout<<“An Exception was thrown”<<endl;
   // deallocate resource here, then rethrow
   throw;
}
```

# 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**.
  - When you call a function, the address where the logic should return at the end of the function is stored in a memory location called the stack.



# Unwinding the stack (continue)



```
int functionB() {
    int ret;
    …
    throw(errorCode1);
    …
    return ret;
}
int functionA() {
    int tmp;
    …
    functionB();
    …
    throw(errorCode2);
    …
    return tmp;
}
```

```
int main() {
    try {
        functionA();
        …
    } catch (exception& err) {
        …
    }
    return 1;
}
```

# 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.

- An alternative to namespaces, to avoid name duplication, is to give functions, classes, or whatever, long names.

  ```
  string cplusplus _primer_make_plural(size_t, string&);
  ```

  But this can be clumsy.

- Namespaces are likely to be more useful for large projects, you likely wouldn’t need them for assignments, but they may be useful for your final project or if you are making libraries of reusable code.

# Syntax …

- Namespaces appear to be defined in a similar way to classes, with a different keyword though.
- To declare a name space of our own we would typically do the following:

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

- A namespace can be defined over multiple files, as the `std` namespace is.
  - This is the property of being discontiguous or open, unlike classes which are closed.

# Using `using`

- 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

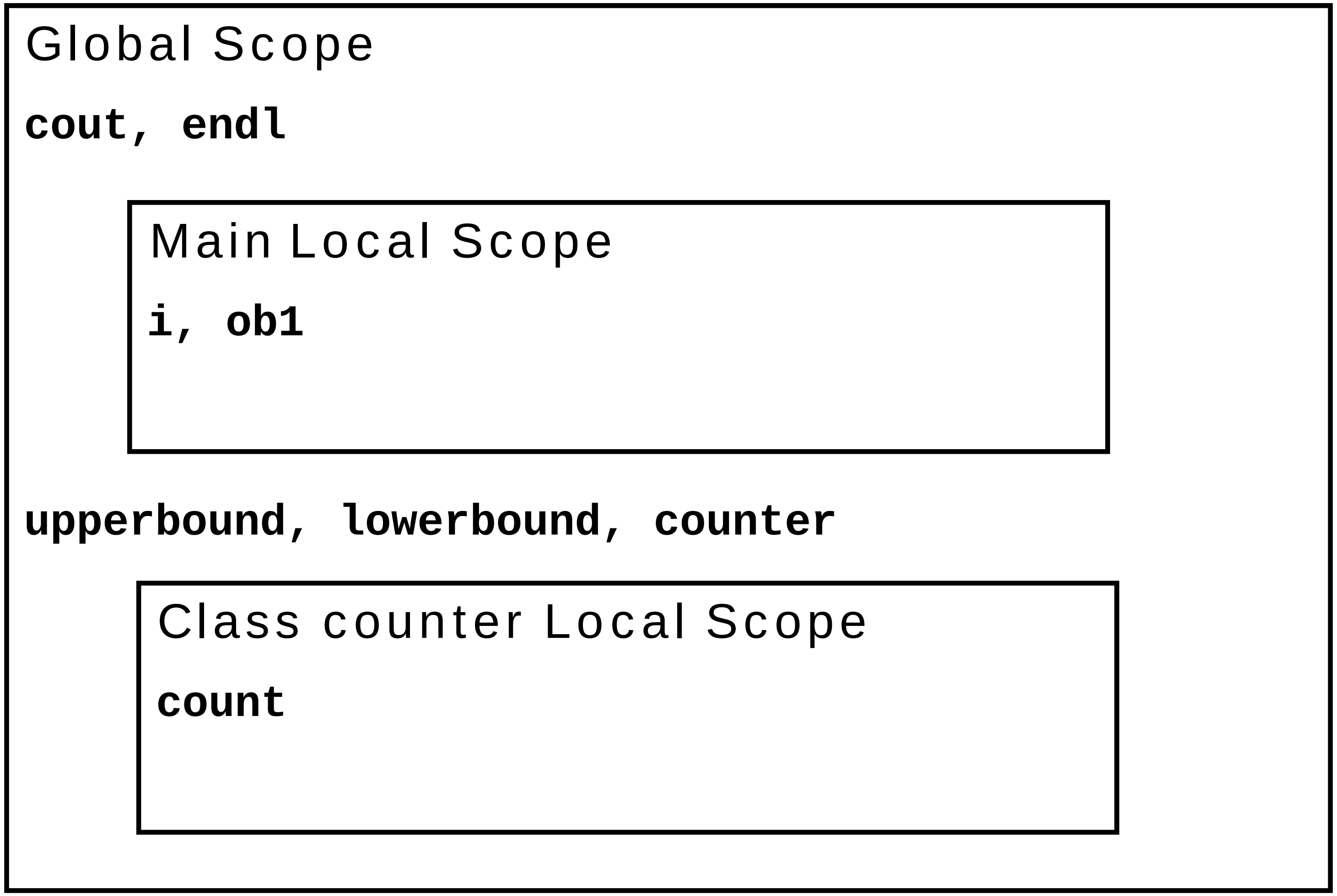
```
#include <iostream>
#include "counter.h"
using namespace std;

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.



Global Scope

**cout, endl**

Main Local Scope

**i, ob1**

**upperbound, lowerbound, counter**

Class counter Local Scope

**count**

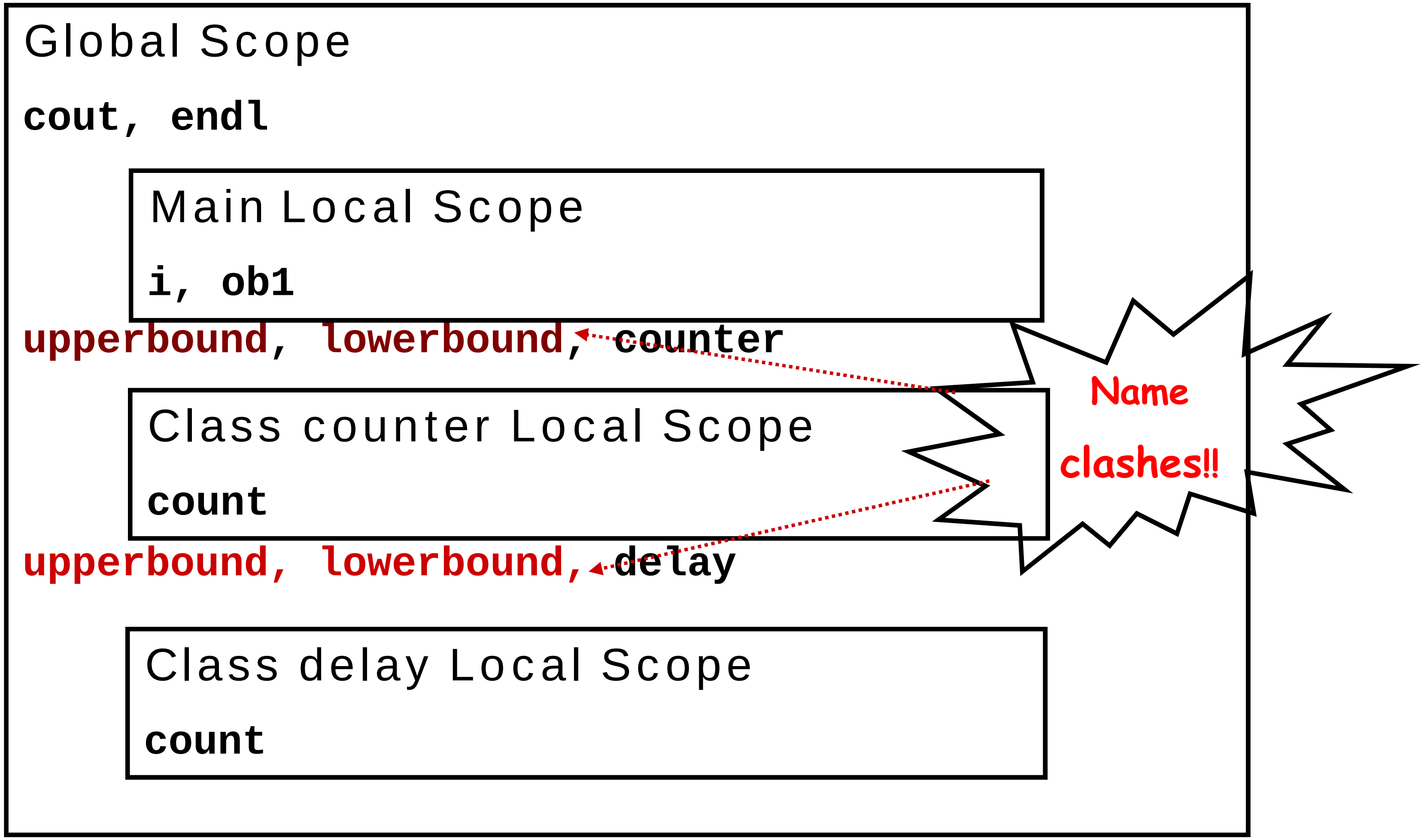
```
#include <iostream>
using namespace std;
#include "counter.h"
#include "delay.h"
int main()
{
    ...;
}
```

delay.h

```
#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

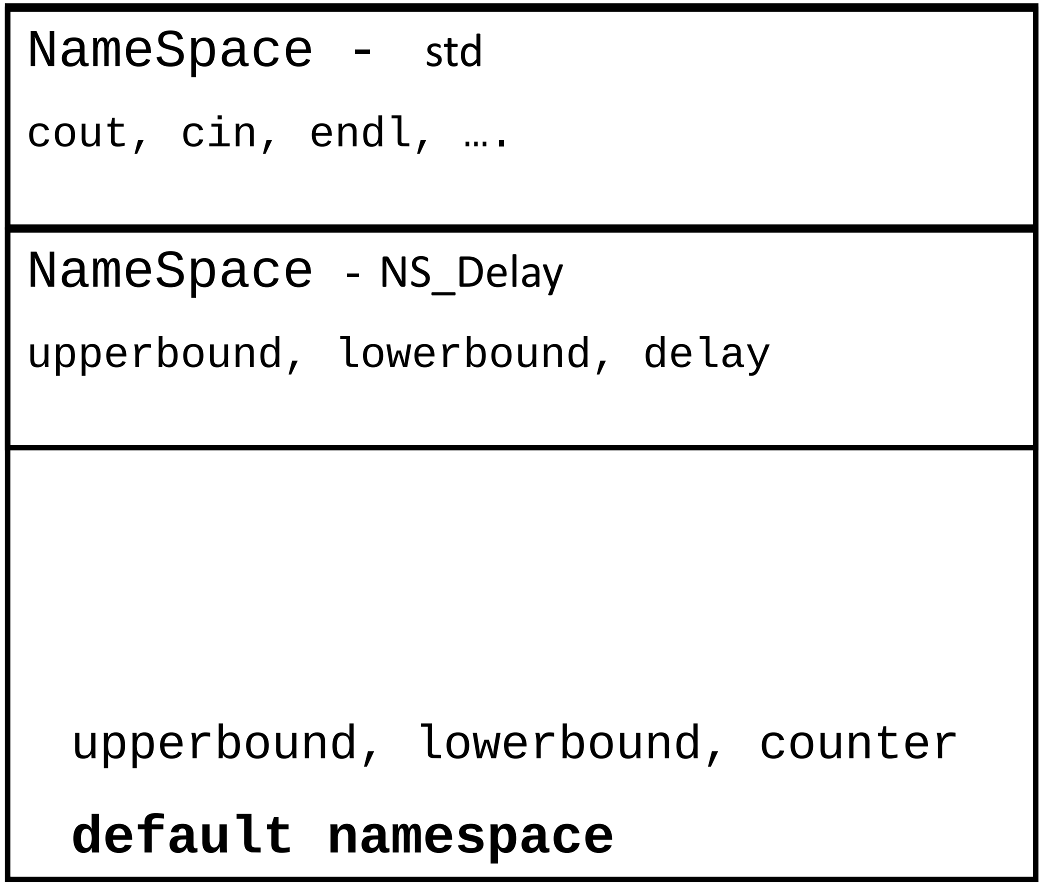
Class delay Local Scope

count

Name clashes!!

- We avoid the clash as follows:

Global Scope



NameSpace - std

cout, cin, endl, ….

NameSpace - NS_Delay

upperbound, lowerbound, delay

upperbound, lowerbound, counter

**default namespace**

std::cout

std::cin

NS_Delay::upperbound

NS_Delay::lowerbound

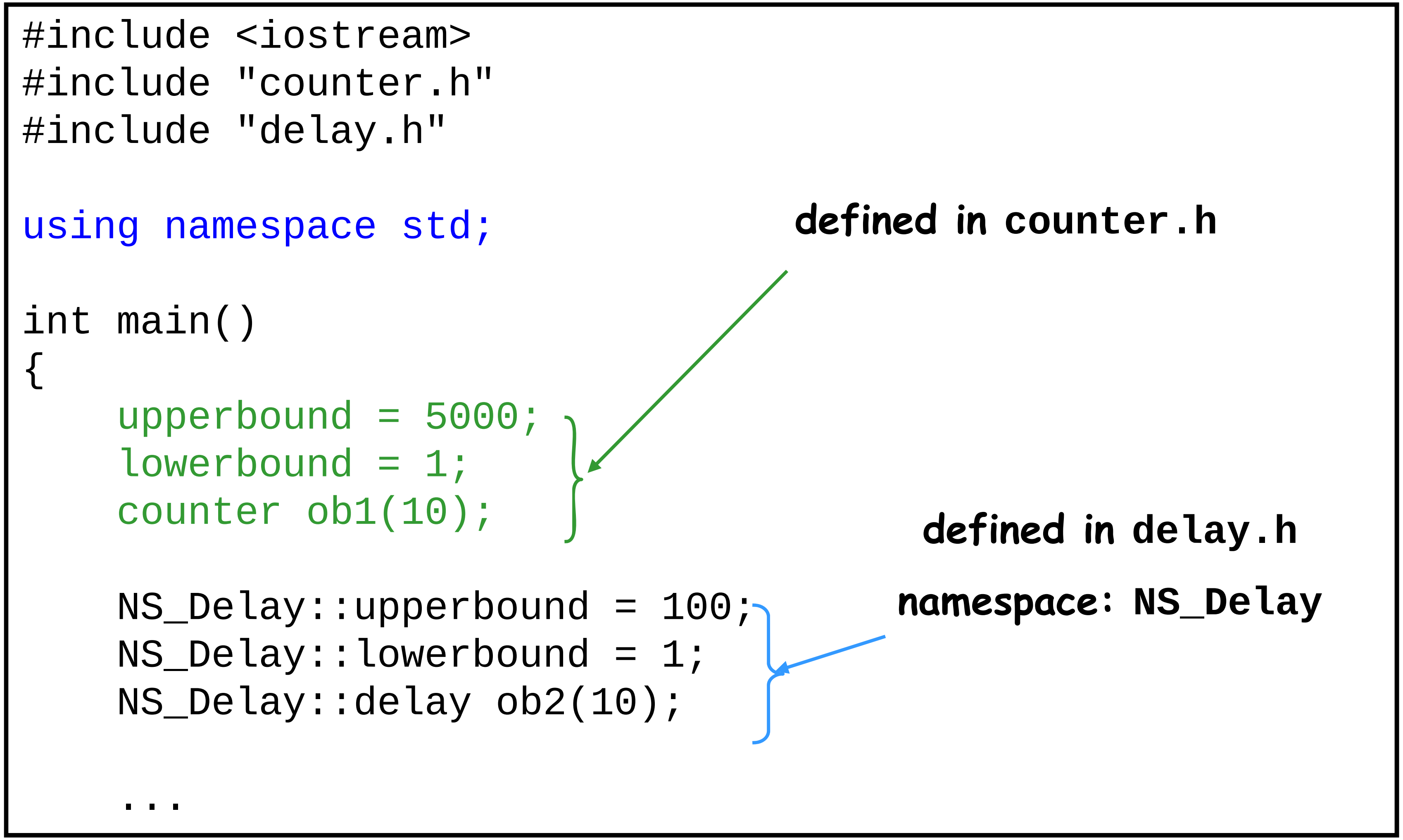
from "delay.h"

upperbound

lowerbound

from "counter.h"

```
#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.
- Inside the namespace, any identifier declared within that namespace can be referred to directly, without any namespace qualification:

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

Those variables are within scope in the namespace.

- However, since namespace defines a scope, you need to use the scope resolution operator to refer to objects declared within a namespace from outside that namespace
- For example, 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;
  ```

# Don’t …

- 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
}
```

- Unnamed namespaces allow you to establish unique identifiers that are known only 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 may be used directly, without qualification.
  - But outside the file, the identifiers are unknown.

# 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:

- These are new to C++11, and they are a type of nested namespace.
- Names in an inline namespace can be used directly in the enclosing namespace.
- The keyword `inline` needs to be used in the first part of the namespace declaration.

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

- The particular use of this the textbook gives is for managing different versions of the textbook code
...

```
namespace cplusplus_primer {
#include “FifthEd.h”
#include “FourthEd.h”
}
```

- By making the FifthEd namespace inline …

```
inline namespace FifthEd{ … }
```

- … the fifth editions of fourth edition functions are directly usable in the cplusplus_primer namespace.

# 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;
```

# Programming defensively …

- You should aim to become familiar with the vulnerabilities of whatever language(s) you are using …
- Here goes a useful reference for C++ …
- http://cwe.mitre.org/data/definitions/659.html

This may still be in CSCI262/CSCI862 too

# Defensive programming

- "The whole point of defensive programming is guarding against errors you don't expect".

  **Steve McConnell, Code Complete**

- The term seems to date back to the 1st edition of Kernighan and Ritchie (The C Programming Language).
  - Referred to both resilience in the presence of bugs, which is what it's mostly interpreted as now, and ...
  - ... Reducing the likelihood of introducing bugs with changes in code.

# Handling errors?

- There is some confusion about the difference between defensive programming and handling errors or exceptions.
- Error handling, exceptions and so on, are about handling errors that are known about and could happen.
  - Bad input and so on. It’s being defensive.
- Some sources say defensive programming is about handling things that shouldn’t happen.
  - Effectively things that cannot happen unless something goes wrong ← A bug.

- From that viewpoint defensive programming is protecting the programmer against themselves ☺
  - If you have a public function that can take data from the user, and the data is supposed to be integers, then testing for non-integer data would be exception handling.
  - If the same function was private and you controlled the method of calling, putting in code to check the passed value would be a defensive programming technique.
- Other sources suggest defensive code is “dealing with exceptions”.

# So what is defensive programming?

- There doesn’t seem to be a single globally accepted definition, but it certainly is supposed to be capture the spirit of secure and reliable programming.
- Liskov, identifies one fairly common characteristic, at least;
  - “. . . defensive programming; that is, writing each procedure to defend itself against errors.”
- Maguire defines “Defensive Programming purely as a way of defending a procedure from crashing, even if this means that the procedure does not perform correctly.”
  - This sometimes means bug hiding.

# What might it include?

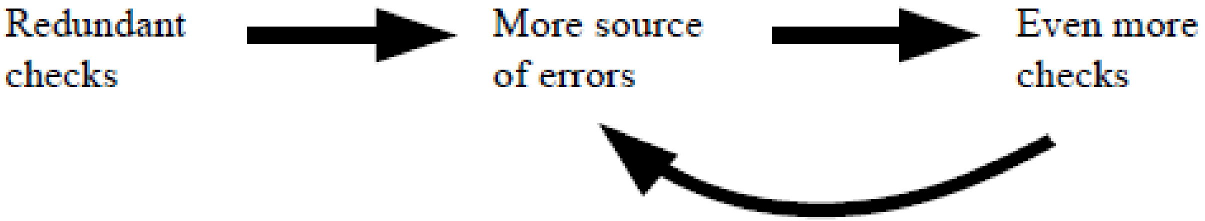
- This list is taken from the Master’s thesis of Roger Andersson and Patrick Jungner, which compared the performance of Defensive Programming and Programming by Contract (Formal agreement between a class and it’s clients regarding the requirements).
  - Consistent indentation makes the source code easier to read and debug.
  - Do not use the default target in a switch-statement to handle a real case. Have cases for every valid value and throw an exception in the default case.
  - “If It Can't Happen, Use Assertions to Ensure It Won't“
  - Program modules should be as independent as possible.

- Program modules should be as independent as possible.
  - Validate all parameters in methods.
  - Validate all return values from methods and system calls.
  - Use meaningful error messages.
  - ……
- They note this list is not exhaustive.
- In the thesis the guiding principle behind defensive programming is taken to be “Check every assumption.”
- And a definition is given as:
  - “A software development principle aiming to increase software quality, by making every method responsible for its own quality.”

# A switch default example

- A function to roll a standard 6-sided die and do something different for each outcome…
- Switching on the value…
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - Default (in the switch keyword sense). Captures the what shouldn’t happen case …

- Meyer’s definition of defensive programming:
  - “A technique of fighting potential errors by making every module check for many possible consistency conditions, even if this causes redundancy of checks performed by clients and suppliers. Contradicts Design by Contract.”
- ... Leads to one of the problems with Defensive Programming (Figure 3.8 from the thesis) ...



# Offensive programming – Fail Fast

- Offensive programming is when bad data results in an error report and an immediate stop → Fail Fast.
  - Assert statements cause termination so could be used as part of this.
- What strategy is appropriate?
  - It depends on the context.

# http://www.defprogramming.com/

- Quotes about coding …
- “When debugging, novices insert corrective code; experts remove defective code.”

  Richard Pattis on debugging.

- “Sometimes it pays to stay in bed on Monday, rather than spending the rest of the week debugging Monday's code.”

  Christopher Thompson on debugging.

- A good programmer is someone who always looks both ways before crossing a one-way street.

  Doug Linder on programmers.