Maciej Sobieraj

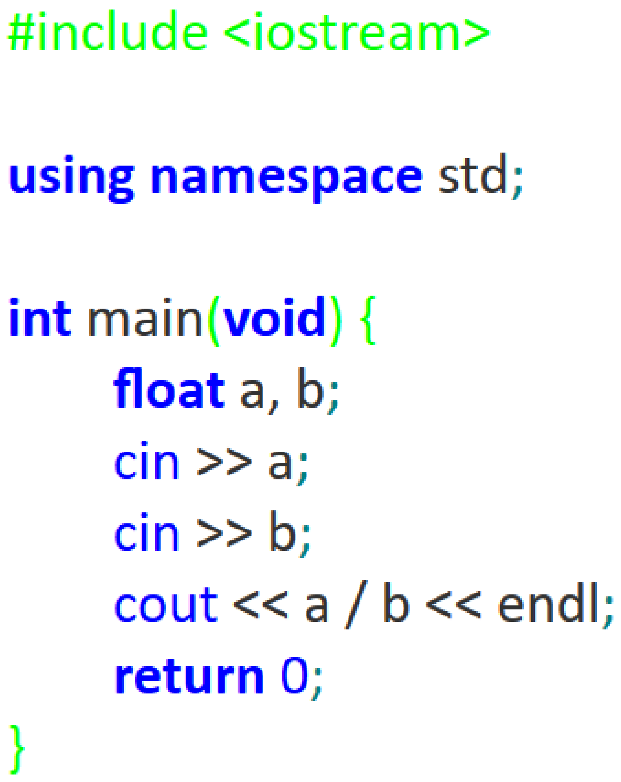
Lecture 7

Outline

1. **Exceptions**
    1. **To err is human**
    2. Throw statement in detail
    3. Categorizing exceptions

How to get into trouble?

- **The code is aborted, as division by zero isn't possible in a world of real numbers**.



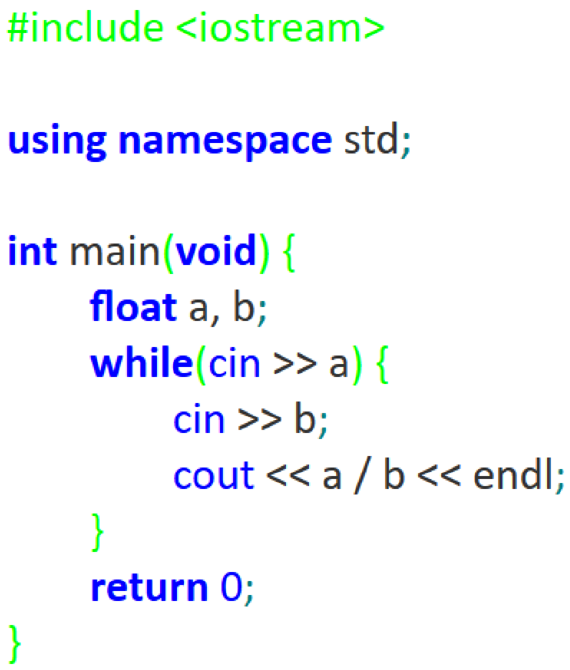
```
#include <iostream>

using namespace std;

int main(void) {
        float a, b;
        cin >> a;
        cin >> b;
        cout << a / b << endl;
        return 0;
}
```

How to get into trouble?

- There is one exception – **if there is no data in the input stream or the data is invalid, the *cin* stream returns a null reference**, which evaluates to false in Boolean contexts.

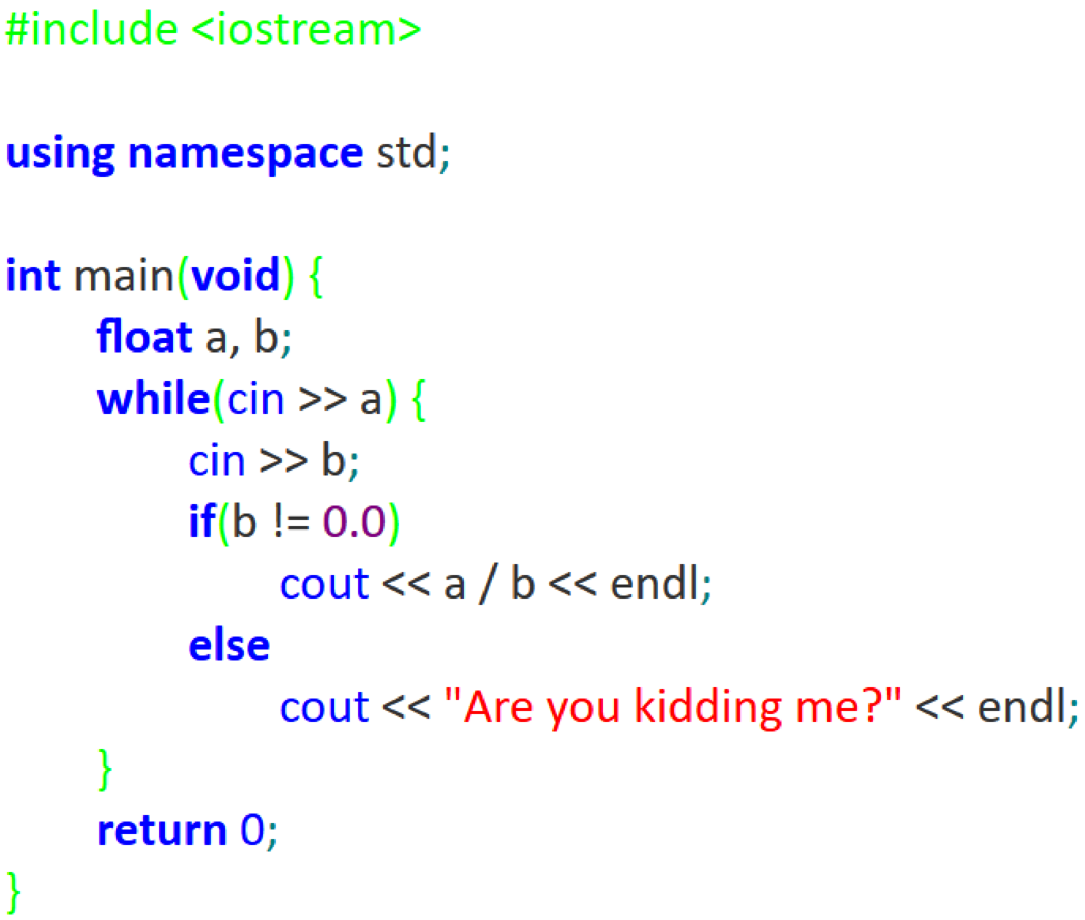


```
#include <iostream>

using namespace std;

int main(void) {
	float a, b;
	while(cin >> a) {
		cin >> b;
		cout << a / b << endl;
	}
	return 0;
}
```

How to get into trouble?



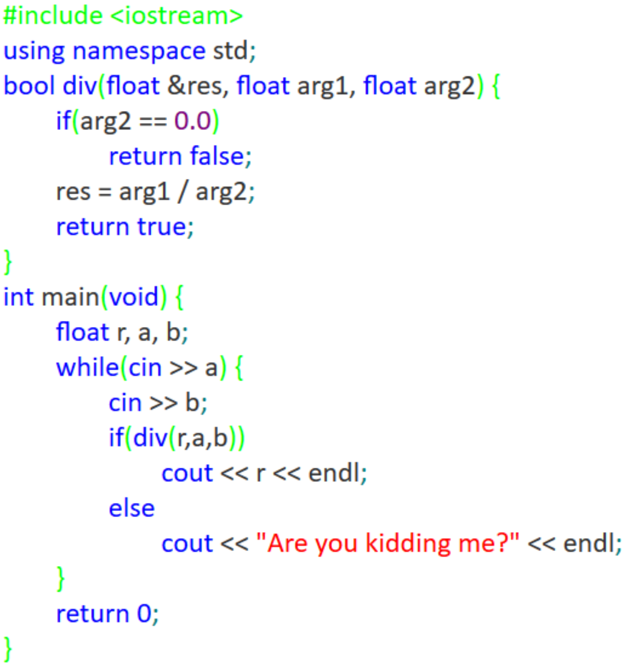
```
#include <iostream>

using namespace std;

int main(void) {
    float a, b;
    while(cin >> a) {
        cin >> b;
        if(b != 0.0)
            cout << a / b << endl;
        else
            cout << "Are you kidding me?" << endl;
    }
    return 0;
}
```

How to get into trouble?

- The idea is simple: **when the function discovers that there’s a problem** with the arguments or with the intermediate results, **it exits immediately**, **returning false as the result**.

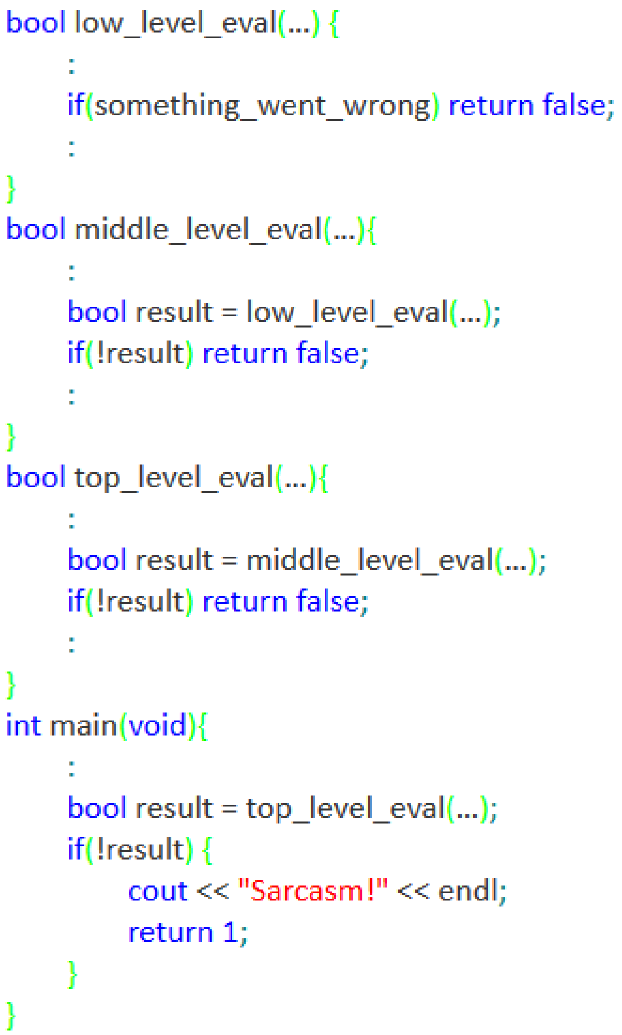


```
#include <iostream>
using namespace std;
bool div(float &res, float arg1, float arg2) {
    if(arg2 == 0.0)
        return false;
    res = arg1 / arg2;
    return true;
}
int main(void) {
    float r, a, b;
    while(cin >> a) {
        cin >> b;
        if(div(r,a,b))
            cout << r << endl;
        else
            cout << "Are you kidding me?" << endl;
    }
    return 0;
}
```

How to get into trouble?

- Try to imagine that our (safe) function **is invoked many times by other functions**.
- Notice that the chain of invoking-invoked functions can be very long. If only the highest-level functions are responsible for reacting to errors occurring on the lower levels, **it may result in the code "swelling"**.
- The swell contains the code that does nothing but discover errors and try to handle them.

How to get into trouble?



```
bool low_level_eval(...) {
        :
        if(something_went_wrong) return false;
        :
}
bool middle_level_eval(...){
        :
        bool result = low_level_eval(...);
        if(!result) return false;
        :
}
bool top_level_eval(...){
        :
        bool result = middle_level_eval(...);
        if(!result) return false;
        :
}
int main(void){
        :
        bool result = top_level_eval(...);
        if(!result) {
                cout << "Sarcasm!" << endl;
                return 1;
        }
}
```

How to get into trouble?

- **An exception is data**.
- Imagine an exception as a winged box, capable of flying, which comes up when something bad occurs, at the time when it happens.
- The box contains data which may help to identify the reason for the failure.
- The data may be of any type: it may be an *int*, a *float*, a string, an object of any class, you name it.

How to get into trouble?

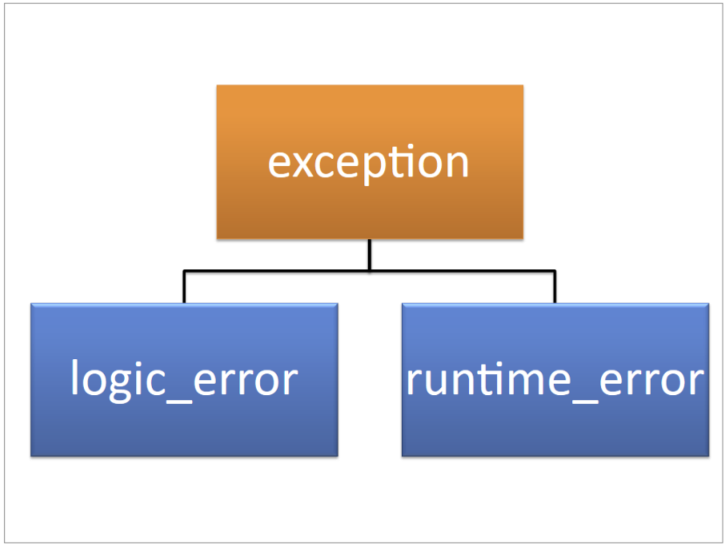
- **The part of the code that may cause problems needs to be marked** (actually nested) within a special kind of block. The block is intended to be carefully watched during its execution.
- **When an exception arises, the execution of the block is terminated**, but the program itself is still alive.
- **The exception is caught by another part of the code.**

How to get into trouble?

- The top-most class, named *exception*, is a base for all exceptions. The class is only used to define some behaviours and properties common to all exceptions.
- Two different classes are derived from the exception class.
  - The first, named *logic_error*, is intended to represent exceptions connected to program logic i.e. the algorithm, its implementation, data validity and cohesion.
  - The second class, named *runtime_error*, is used to identify exceptions thrown due to “unexpected” accidents like a lack of memory.

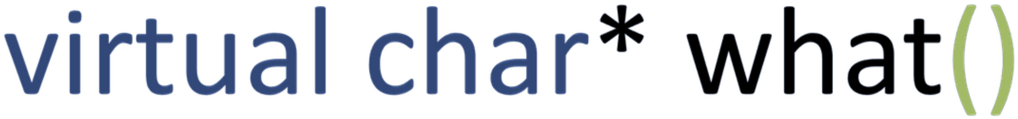
How to get into trouble?

- All these entities are defined within the header file *exception*. This means that the line
  - #include <exception>
- may be needed in a code which makes use of any of these classes.



Anatomy of an exception object

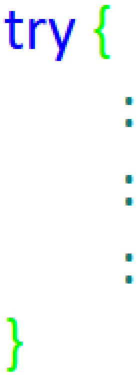
- The exception class is very modest. In fact it defines only three components:
  - a constructor (not very useful to us because, as we’ve mentioned before, objects of this class aren’t created
  - a virtual destructor, originally empty
  - a virtual function called *what* which returns the C-style string (a pointer to the array of characters terminated by the **null** (‘\0’) character



```
virtual char* what()
```

Where are exceptions thrown?

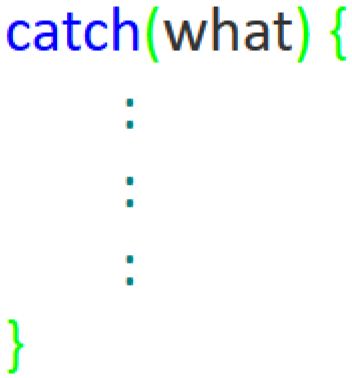
- If you want to detect exceptions, you need to mark the part of the code in which the exceptions may occur.
- You do this by using the “try” statement.



```
try {
        :
        :
        :
}
```

Where are exceptions caught?

- If you’re determined to catch any of the flying exceptions, you need to put the *catch* statement directly after the *try* statement.



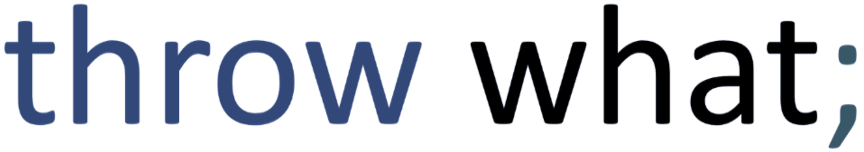
```
catch(what) {
        :
        :
        :
}
```

Where are exceptions caught?

- If you write it this way:
  - catch(string &anyproblem) { … }
- it'll mean: *I want to catch the exceptions which carry strings*.
- This form:
  - catch(exception &otherproblem) { … }
- means: *I'm going to catch the exceptions carrying objects of the exception class or of any other classes derived from the exception cla*

How are exceptions thrown?

- If you want to throw an exception, you have to use the statement of the same name. The *throw* statement requires data that’ll be “packed” into an exception before its departure.



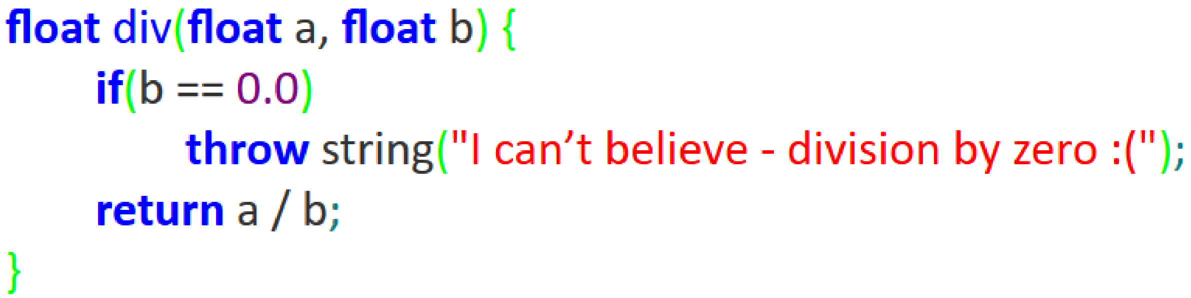
```
throw what;
```

How are exceptions thrown?

- f you’re going to dispatch an *int* value, you’ll write something like this:
  - throw 997;
- If you want to throw an exception equipped with an object of any class, you need to specify the constructor to be invoked to prepare the data, like this:
  - throw string("Bye world!");

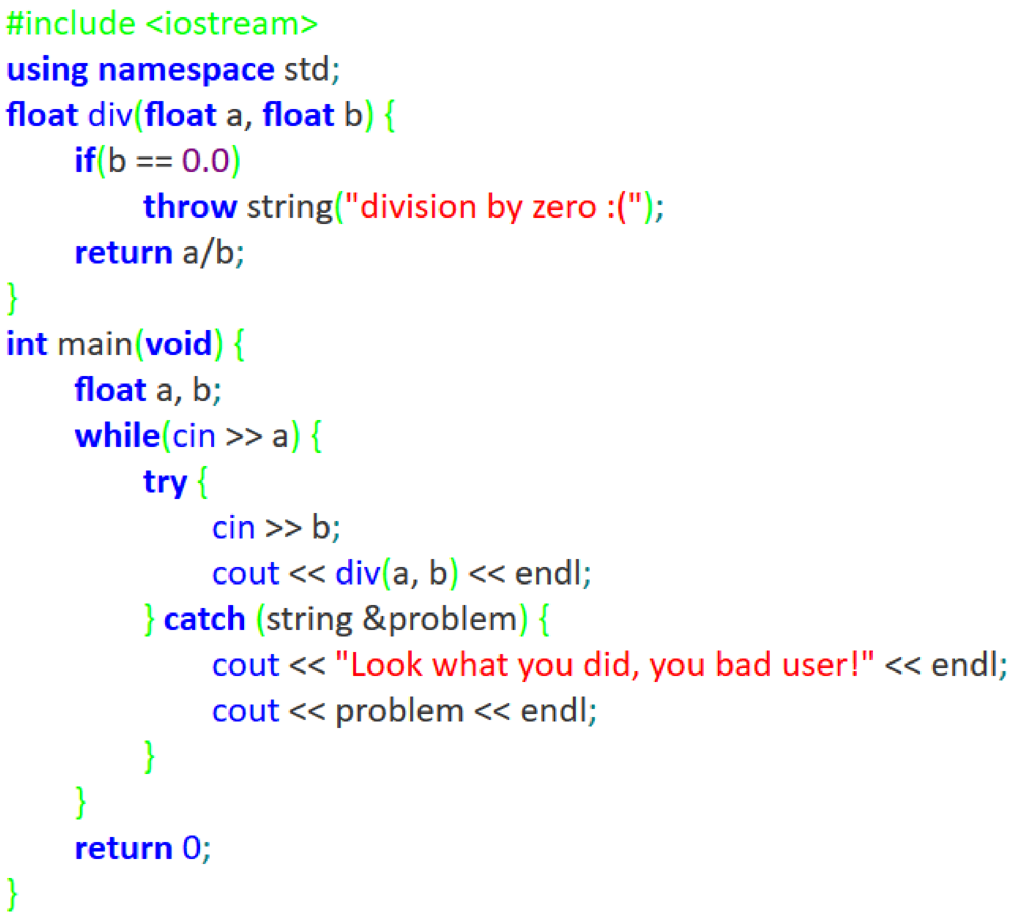
How are exceptions thrown?

- The function will provide a result if the arguments are valid, otherwise it’ll throw an exception containing a complaining string.



```
float div(float a, float b) {
        if(b == 0.0)
                throw string("I can’t believe - division by zero :(");
        return a / b;
}
```

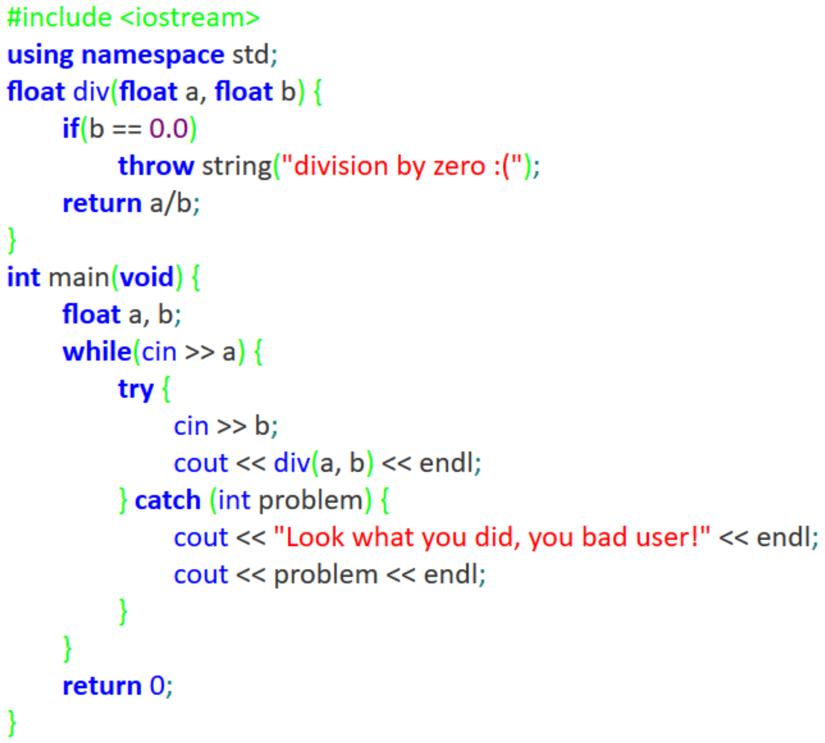
How are exceptions thrown?



```
#include <iostream>
using namespace std;
float div(float a, float b) {
    if(b == 0.0)
        throw string("division by zero :(");
    return a/b;
}
int main(void) {
    float a, b;
    while(cin >> a) {
        try {
            cin >> b;
            cout << div(a, b) << endl;
        } catch (string &problem) {
            cout << "Look what you did, you bad user!" << endl;
            cout << problem << endl;
        }
    }
    return 0;
}
```

How are exceptions thrown?

- The program will be aborted with a message saying that an instance of an unhandled exception has been thrown.

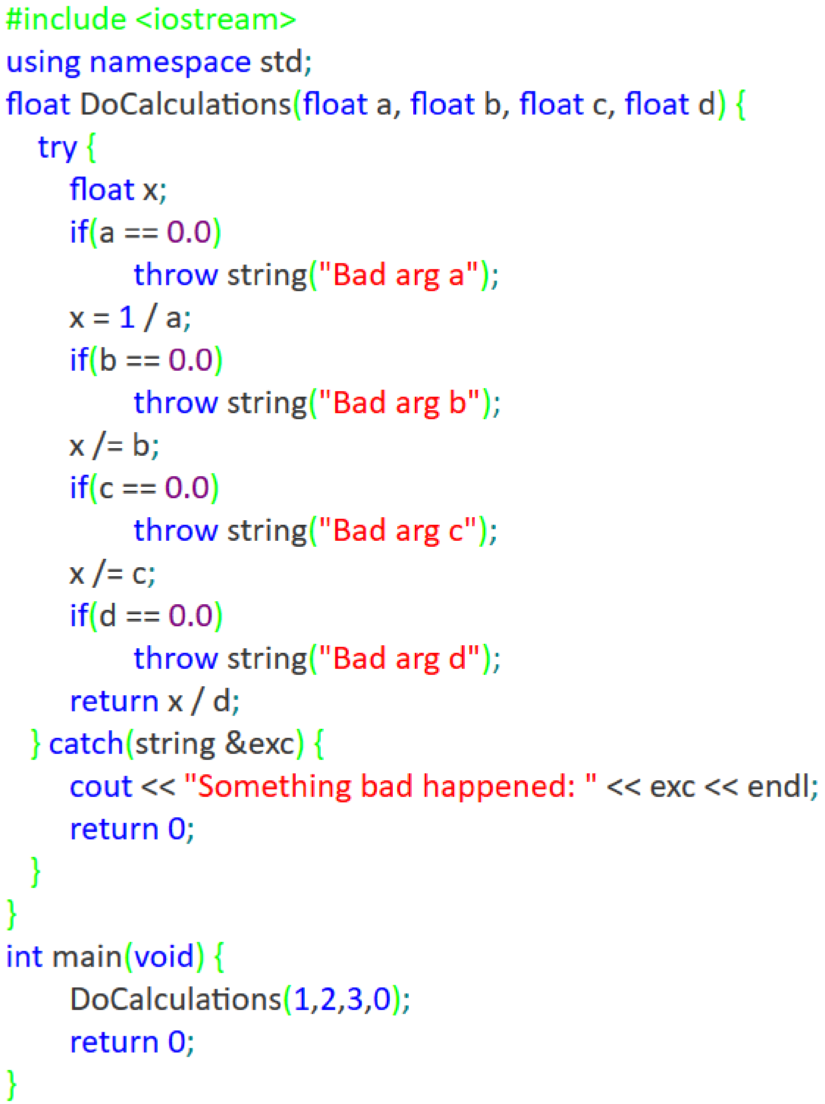


```
#include <iostream>
using namespace std;
float div(float a, float b) {
    if(b == 0.0)
        throw string("division by zero :(");
    return a/b;
}
int main(void) {
    float a, b;
    while(cin >> a) {
        try {
            cin >> b;
            cout << div(a, b) << endl;
        } catch (int problem) {
            cout << "Look what you did, you bad user!" << endl;
            cout << problem << endl;
        }
    }
    return 0;
}
```

Outline

1. **Exceptions**
   1. To err is human
   2. **Throw statement in detail**
   3. Categorizing exceptions

Throw and catch coupled together



```
#include <iostream>
using namespace std;
float DoCalculations(float a, float b, float c, float d) {
    try {
        float x;
        if(a == 0.0)
                throw string("Bad arg a");
        x = 1 / a;
        if(b == 0.0)
                throw string("Bad arg b");
        x /= b;
        if(c == 0.0)
                throw string("Bad arg c");
        x /= c;
        if(d == 0.0)
                throw string("Bad arg d");
        return x / d;
   } catch(string &exc) {
        cout << "Something bad happened: " << exc << endl;
        return 0;
   }
}
int main(void) {
        DoCalculations(1,2,3,0);
        return 0;
}
```

Throw and catch coupled together

- The example program outputs the following text to the screen:
  - Something bad happened: Bad arg d

Throw and catch coupled together

- The exception specification placed in the *catch* branch header, e.g. this one:
  - **catch**(string &exc)
- **works like a local (automatic) variable declaration**.

Throw and catch coupled together

- Inside the following snippet:

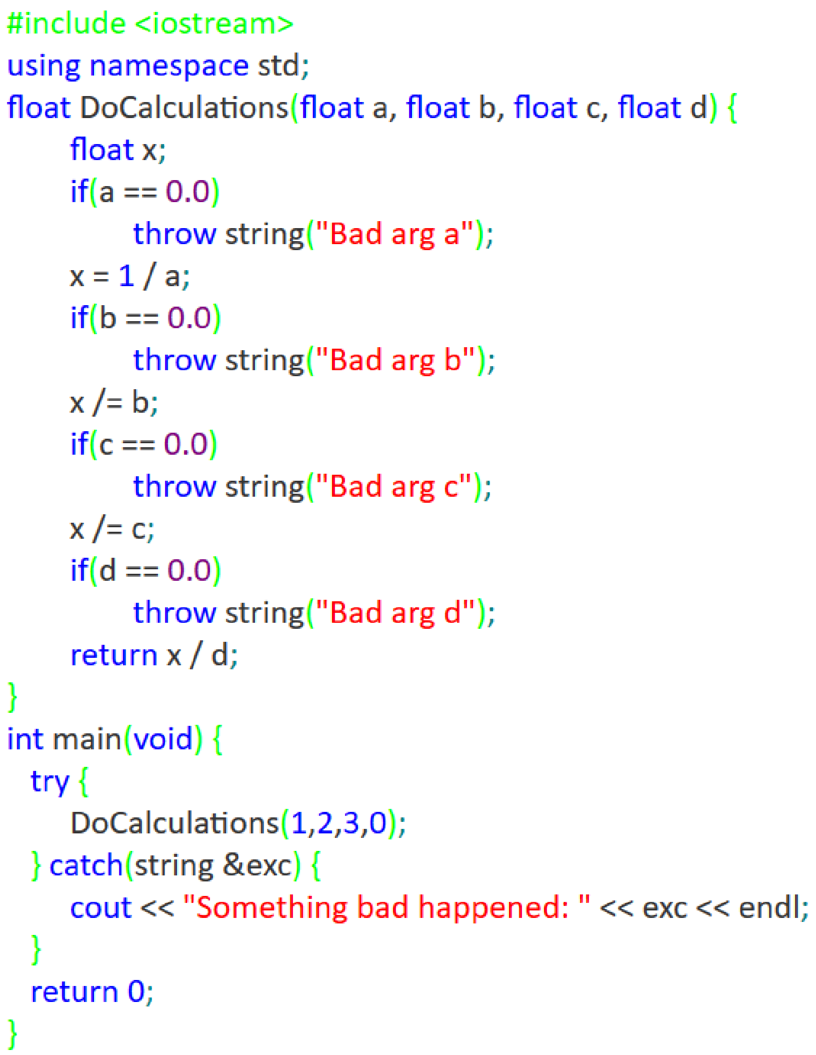
```
int main(void) {
string str;
    try {
        throw string("1");
    } catch(string &str) {
        cout << str;
    }
    return 0;
}
```

- there are **two different** variables, named *st* (the former is hidden by the latter inside the *catch* block).

Throw and catch separated

- As you know, the *throw* and the *catch* may live separately as well. We can put *throw* in one function, *catch* in another, and the mechanism will still work effectively, but of course, only when the functions invoke themselves in the proper order.
- This means that the exception object is **able to fly above the function’s boundaries** and can even skip over more than one function in ord to find its own *catch*.

Throw and catch separated

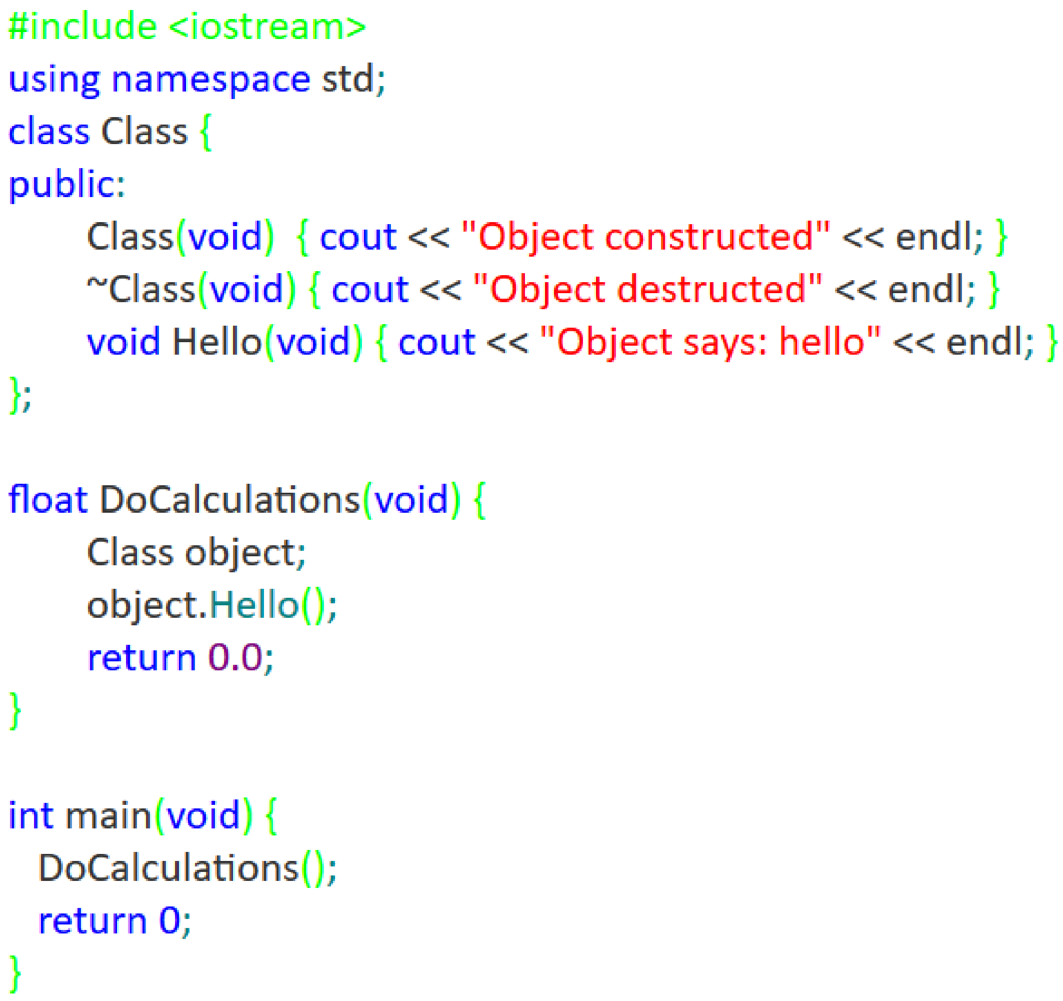


```
#include <iostream>
using namespace std;
float DoCalculations(float a, float b, float c, float d) {
        float x;
        if(a == 0.0)
                throw string("Bad arg a");
        x = 1 / a;
        if(b == 0.0)
                throw string("Bad arg b");
        x /= b;
        if(c == 0.0)
                throw string("Bad arg c");
        x /= c;
        if(d == 0.0)
                throw string("Bad arg d");
        return x / d;
}
int main(void) {
   try {
        DoCalculations(1,2,3,0);
   } catch(string &exc) {
        cout << "Something bad happened: " << exc << endl;
   }
   return 0;
}
```

Throw vs. function epilogue

- The functions executions consist, in general, of three phases:
  - **prologue** (when all automatic variables/objects are created),
  - **execution** (when the function code is performed) and
  - **epilogue** (when the previously created entities are destructed).

Throw vs. function epilogue



```
#include <iostream>
using namespace std;
class Class {
public:
        Class(void)  { cout << "Object constructed" << endl; }
        ~Class(void) { cout << "Object destructed" << endl; }
        void Hello(void) { cout << "Object says: hello" << endl; }
};

float DoCalculations(void) {
        Class object;
        object.Hello();
        return 0.0;
}

int main(void) {
   DoCalculations();
   return 0;
}
```

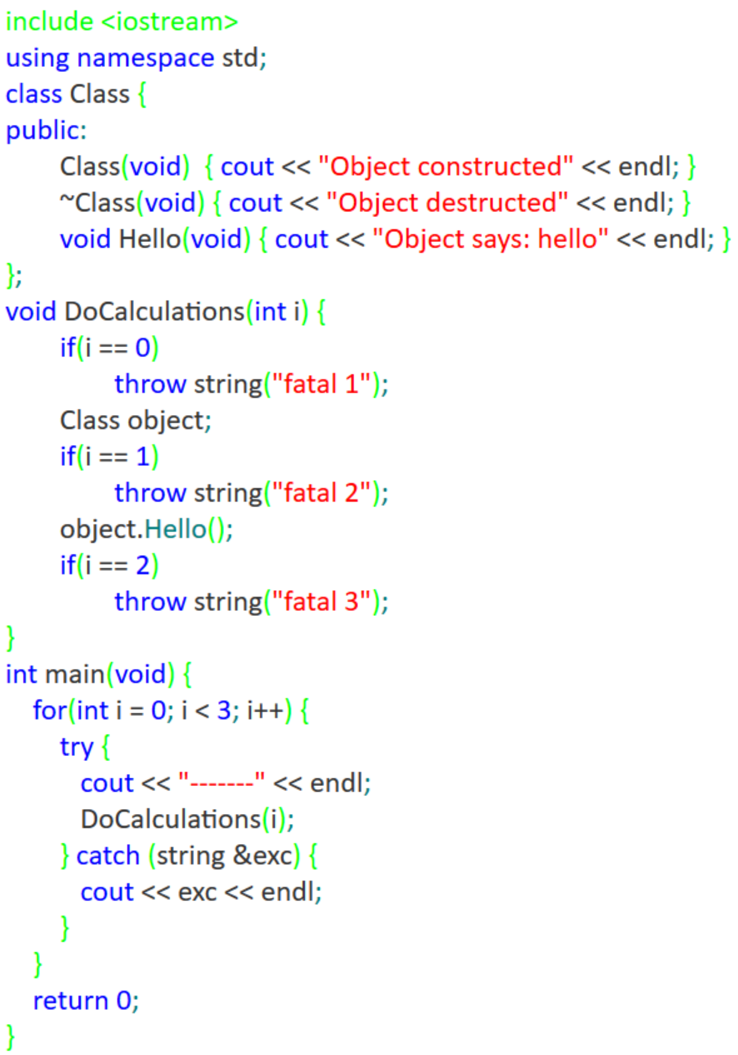
Throw vs. function epilogue

- The program will produce the following output:
  - Object constructed
  - Object says: hello
  - Object destructed

Throw vs. function epilogue

- We’ve added three *throw* instructions within the *DoCalculations* function.
- The Class definition remains the same.
- The *main* function will invoke *DoCalculations* three times and we’ll be able to observe the function’s behaviour.

Throw vs. function epilogue



```
include <iostream>
using namespace std;
class Class {
public:
        Class(void)  { cout << "Object constructed" << endl; }
        ~Class(void) { cout << "Object destructed" << endl; }
        void Hello(void) { cout << "Object says: hello" << endl; }
};
void DoCalculations(int i) {
        if(i == 0)
                throw string("fatal 1");
        Class object;
        if(i == 1)
                throw string("fatal 2");
        object.Hello();
        if(i == 2)
                throw string("fatal 3");
}
int main(void) {
    for(int i = 0; i < 3; i++) {
        try {
           cout << "-------" << endl;
           DoCalculations(i);
        } catch (string &exc) {
           cout << exc << endl;
        }
    }
    return 0;
}
```

Throw vs. function epilogue

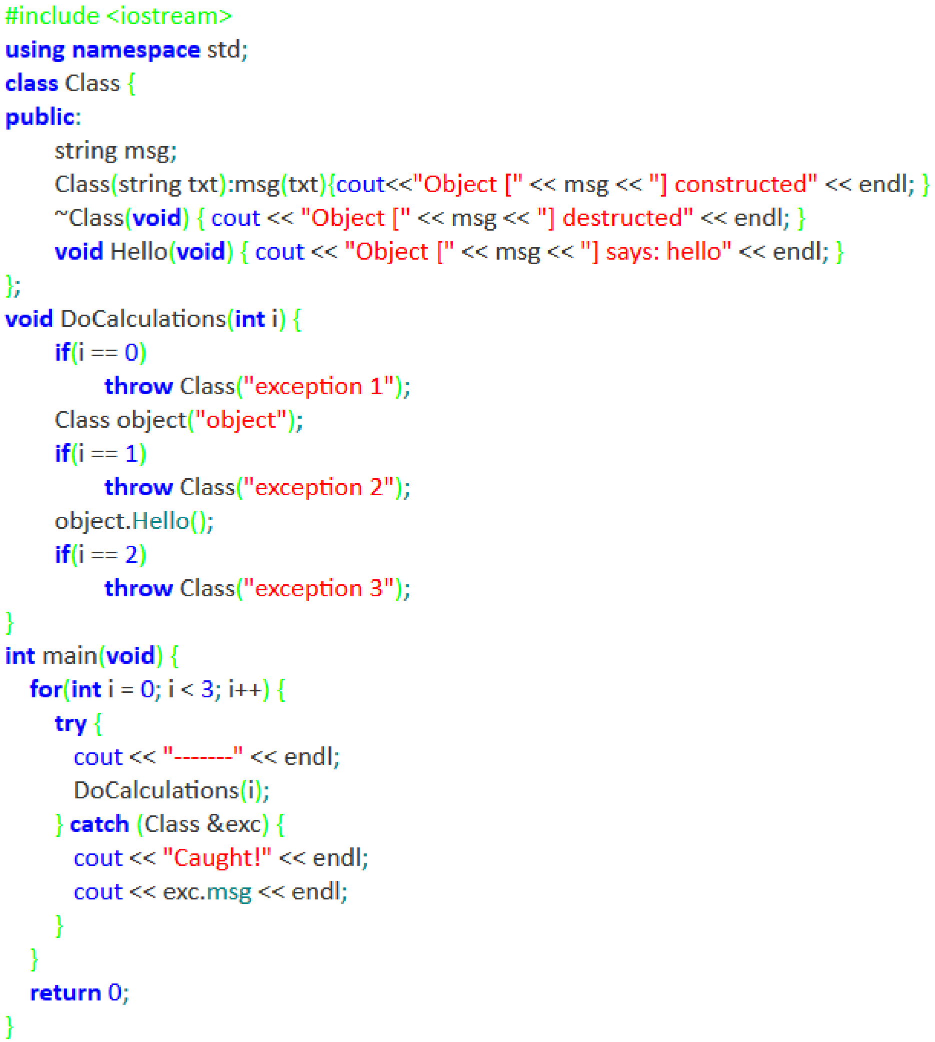
- The program outputs the following text:

```
-------
fatal 1
-------
Object constructed
Object destructed
fatal 2
-------
Object constructed
Object says: hello
Object destructed
fatal 3
```

Throw and the objects it throws

- The ***throw*** **statement is obligated to throw a value** e.g. an object
- *throw* is able to throw **any object of any accessible class**

Throw and the objects it throws



```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt):msg(txt){cout<<"Object [" << msg << "] constructed" << endl; }
        ~Class(void) { cout << "Object [" << msg << "] destructed" << endl; }
        void Hello(void) { cout << "Object [" << msg << "] says: hello" << endl; }
};
void DoCalculations(int i) {
        if(i == 0)
                throw Class("exception 1");
        Class object("object");
        if(i == 1)
                throw Class("exception 2");
        object.Hello();
        if(i == 2)
                throw Class("exception 3");
}
int main(void) {
    for(int i = 0; i < 3; i++) {
        try {
           cout << "-------" << endl;
           DoCalculations(i);
        } catch (Class &exc) {
           cout << "Caught!" << endl;
           cout << exc.msg << endl;
        }
    }
    return 0;
}
```

Throw and the objects it throws

- Be aware that executing a line like this:
  - **throw** Class("exception 1");
- will cause **the creation of a new object** of class Class.
- This means that **the appropriate constructor will be invoked** before the function ends its life.

Throw and the objects it throws

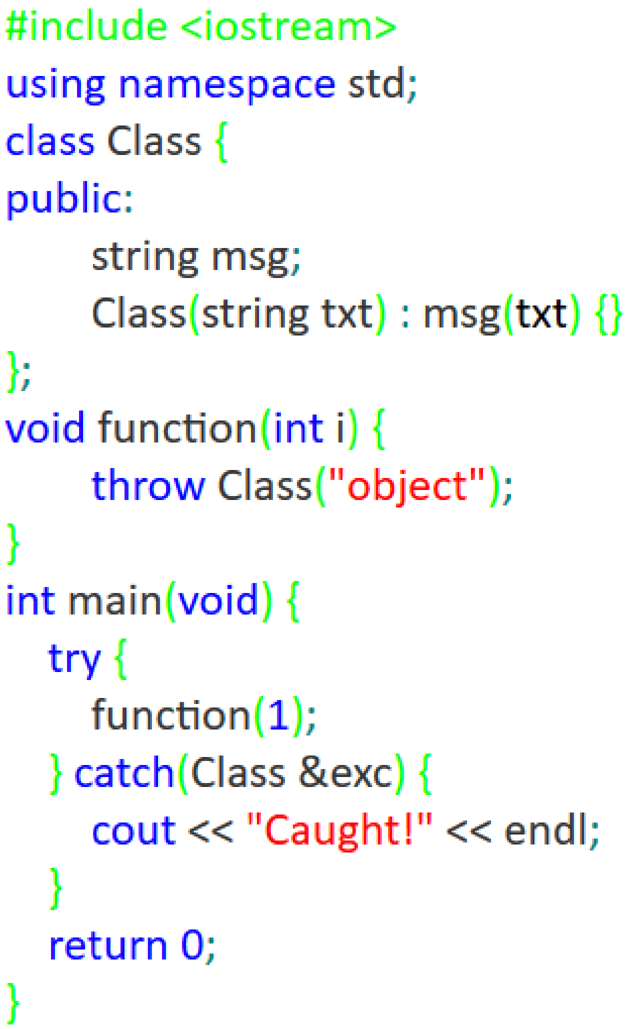
- This program produces the following output:

```
-------
Object [exception 1] constructed
Caught!
exception 1
Object [exception 1] destructed
-------
Object [object] constructed
Object [exception 2] constructed
Object [object] destructed
Caught!
exception 2
Object [exception 2] destructed
-------
Object [object] constructed
Object [object] says: hello
Object [exception 3] constructed
Object [object] destructed
Caught!
exception 3
Object [exception 3] destructed
```

Throw and how we can find out about it

- How can we find out if a function throws any exceptions or not?
- There are two important arguments worth considering:
  - The function may be very long and very complex – reading it may be time consuming and you may overlook some of the throw statements
  - The source code of the function may be inaccessible – it may happen if you use a ready-made library, written by other authors, when you've compiled (binary) files containing only executable code and header files specifying function's headers but not the bodies.

Throw and how we can find out about it



```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(int i) {
        throw Class("object");
}
int main(void) {
    try {
        function(1);
    } catch(Class &exc) {
        cout << "Caught!" << endl;
    }
    return 0;
}
```

Throw and its specification

- **A function, which throws an exception, may (but doesn’t have to) specify the types of the entities being thrown.**
- **It could be used even when the function’s body is inaccessible or hidden**.
- It enables the programmer to announce all exceptions that may leave the function, and therefore prepare other programmers for events that might happen during the function’s execution.

Throw and its specification

- There’s more than one form of specification – the simplest looks like this:
  - throw(x)
- This means that **the function throws one kind of exception**, of type *x*, for example:
  - void function(void) throw(string);

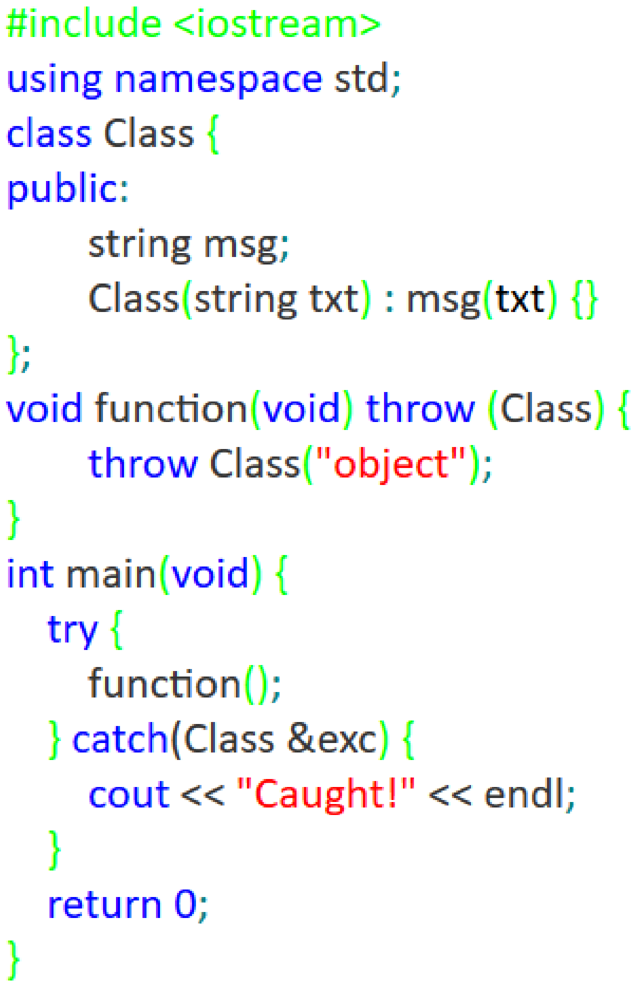
Throw and its specification

- The more complex form looks like this:
  - throw(x1,x2,..,xn)
- This means that the function **throws *n* different exceptions** of types *x1*, *x2*, …, *xn* respectively, for example:
  - int doit(int i) throw(int, string, Class);
- This function may throw exceptions of type *int*, *string* and *Class*.

Throw and its specification

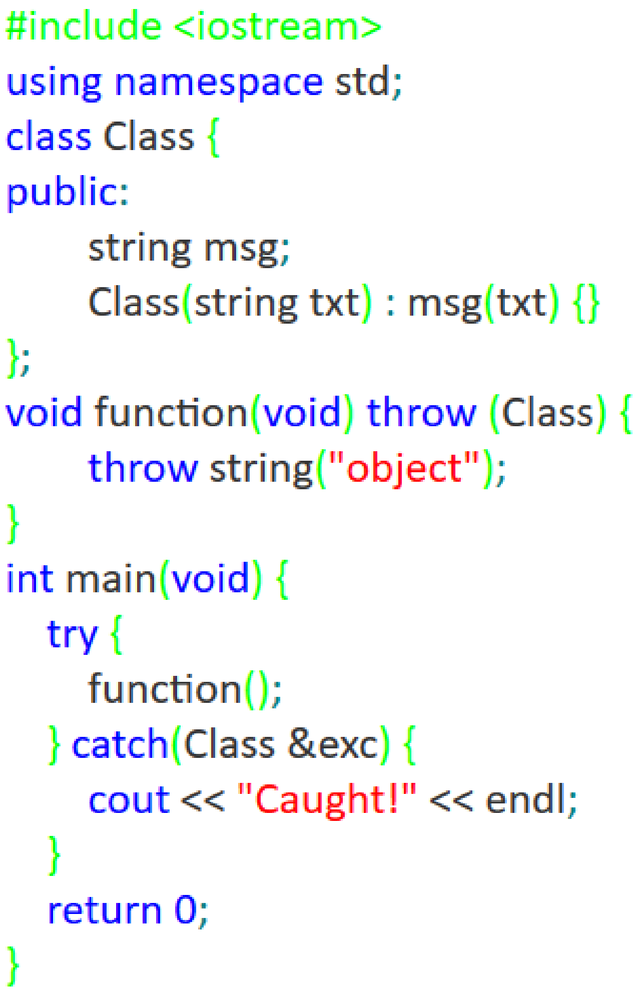
- The last form look like this:
  - throw()
- and means “**the function throws no exceptions at all**”.

Throw and its specification



```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(void) throw (Class) {
        throw Class("object");
}
int main(void) {
    try {
        function();
    } catch(Class &exc) {
        cout << "Caught!" << endl;
    }
    return 0;
}
```

Throw and its specification



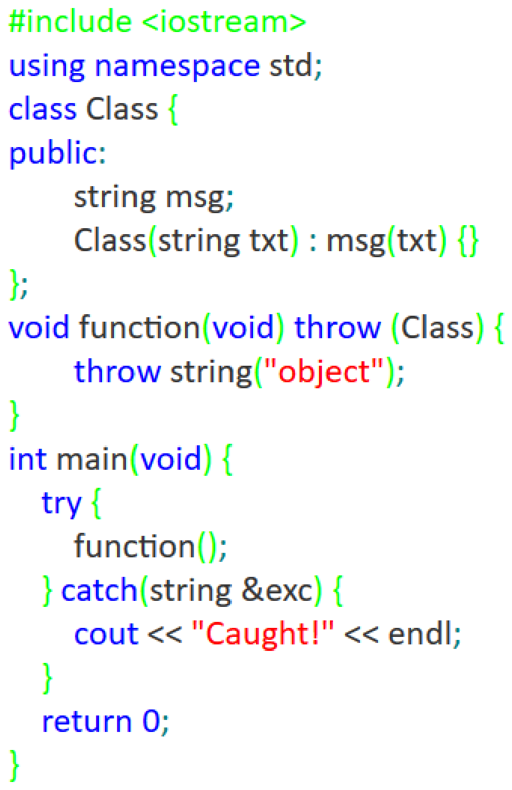
```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(void) throw (Class) {
        throw string("object");
}
int main(void) {
    try {
        function();
    } catch(Class &exc) {
        cout << "Caught!" << endl;
    }
    return 0;
}
```

Throw and its specification

- The function specifies that **it throws exceptions of type *Class***, although **it actually throws *string***.
- What’ll happen then?
- Compilation? **Goes OK** – no problems, no errors, no warnings.
- Execution? Houston, we have a problem – the program’s been **interrupted** and a message has appeared. It says that there was **an uncaught exception of type ‘*std::string*’**.

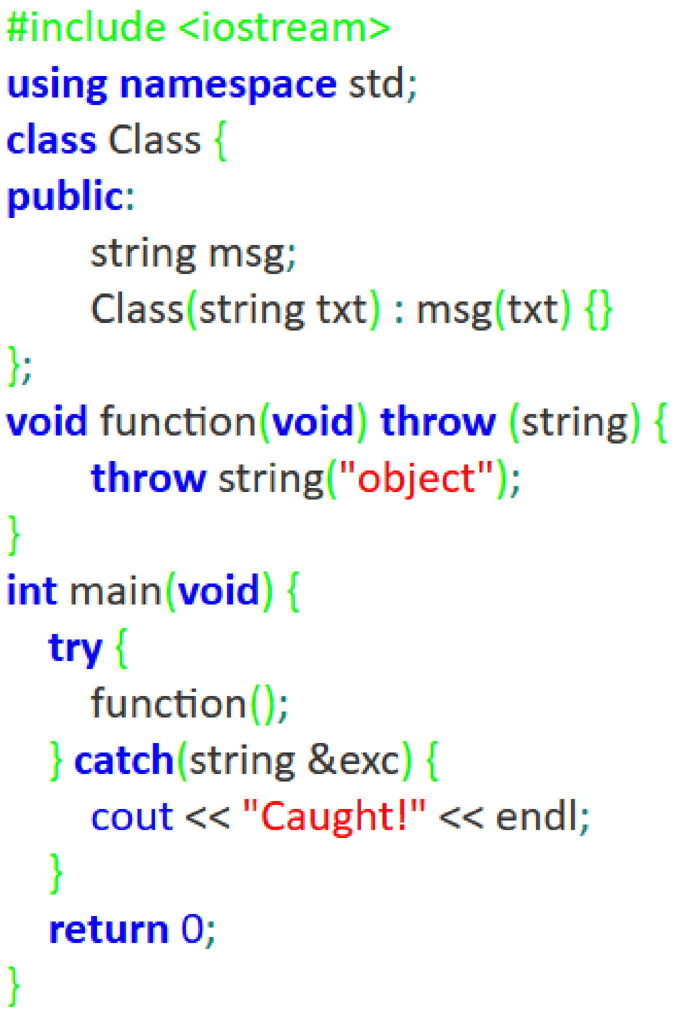
Throw and its specification

- Unfortunately, the correction **hasn’t corrected** our problem at all.



```
#include <iostream>
using namespace std;
class Class {
public:
	string msg;
	Class(string txt) : msg(txt) {}
};
void function(void) throw (Class) {
	throw string("object");
}
int main(void) {
   try {
	function();
   } catch(string &exc) {
	cout << "Caught!" << endl;
   }
   return 0;
}
```

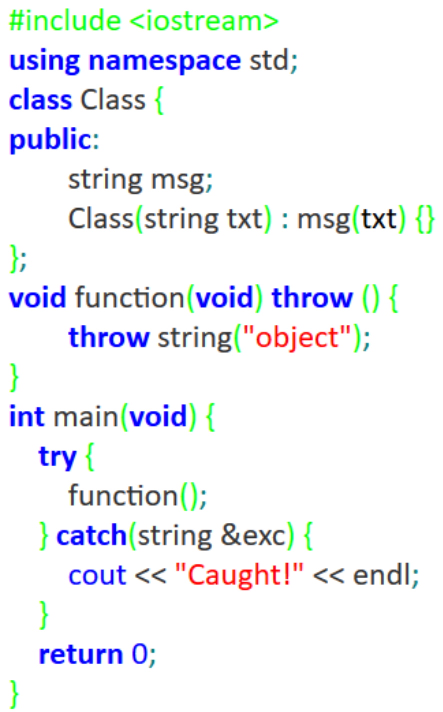
Throw and its specification



```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(void) throw (string) {
        throw string("object");
}
int main(void) {
    try {
        function();
    } catch(string &exc) {
        cout << "Caught!" << endl;
    }
    return 0;
}
```

Throw and its specification

- Are the results as expected?



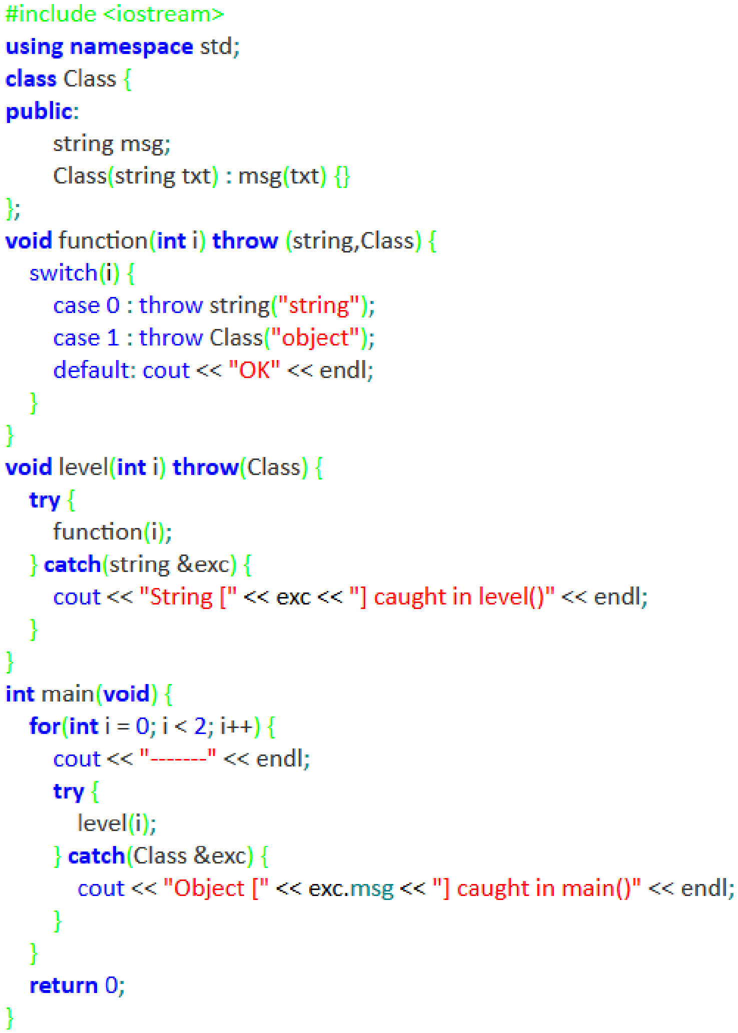
```
#include <iostream>
using namespace std;
class Class {
public:
        string msg;
        Class(string txt) : msg(txt) {}
};
void function(void) throw () {
        throw string("object");
}
int main(void) {
    try {
        function();
    } catch(string &exc) {
        cout << "Caught!" << endl;
    }
    return 0;
}
```

  - Warning: function assumed not to throw an exception but does

Throw and its specification

- **exception handling may be distributed among different parts of the program**.
- You can handle your exceptions in the most suitable places and don’t need to collect all catches in one function or module.

Throw and its specification



```
#include <iostream>
using namespace std;
class Class {
public:
    string msg;
    Class(string txt) : msg(txt) {}
};
void function(int i) throw (string,Class) {
  switch(i) {
    case 0 : throw string("string");
    case 1 : throw Class("object");
    default: cout << "OK" << endl;
  }
}
void level(int i) throw(Class) {
  try {
    function(i);
  } catch(string &exc) {
    cout << "String [" << exc << "] caught in level()" << endl;
  }
}
int main(void) {
  for(int i = 0; i < 2; i++) {
    cout << "-------" << endl;
    try {
      level(i);
    } catch(Class &exc) {
      cout << "Object [" << exc.msg << "] caught in main()" << endl;
    }
  }
  return 0;
}
```

Unexpected exceptions handling

- **If any unexpected exception appears, a special runtime function is invoked**.
- This is the function that **terminates the program and emits the diagnostic message** we’ve read a few times already.
- Its name is *unexpected()*.

Unexpected exceptions handling

- If you really need to do something during the last breaths of the program, you should:
  - code a parameter-less function of type *void*
  - invoke a function called *set_unexpected*, passing the name of your function to it

Unexpected exceptions handling

```
#include <iostream>
#include <string>

using namespace std;
class Class {
public:
      string msg;
      Class(string txt) : msg(txt) {}
};
void function(void) throw () {
      throw string("object");
}
void lastchance(void) {
      cout << "See what you've done! You've thrown an illegal exception!" << endl;
}
int main(void) {
   set_unexpected(lastchance);
   try {
      function();
   } catch(string &exc) {
      cout << "Caught!" << endl;
   }
   return 0;
}
```

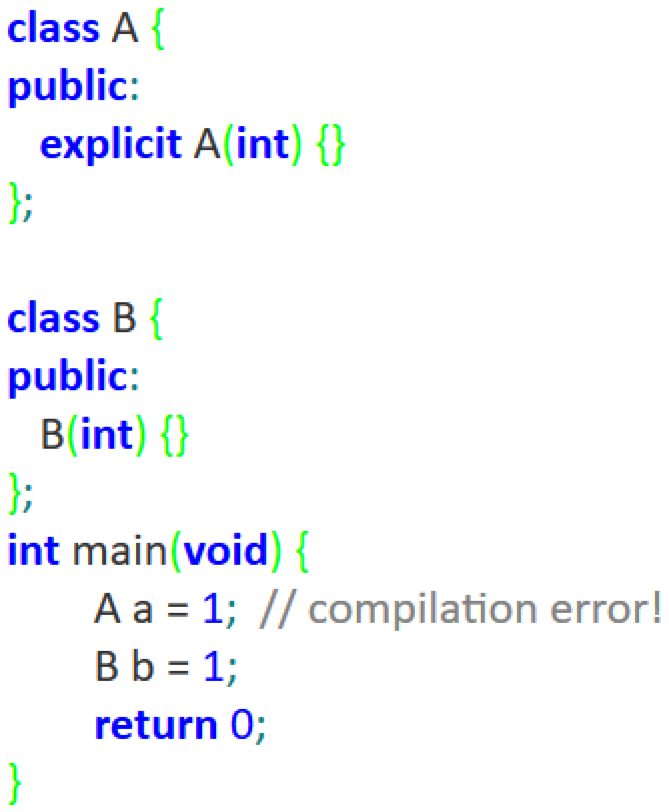
Outline

1. **Exceptions**
    1. To err is human
    2. Throw statement in detail
    3. **Categorizing exceptions**

The ‘explicit’ keyword

- The *explicit* keyword may be placed in front of a class’s constructor declaration.
- It protects the constructor from being used in any context requiring the use of implicit conversions.

The ‘explicit’ keyword



```
class A {
public:
   explicit A(int) {}
};

class B {
public:
   B(int) {}
};
int main(void) {
        A a = 1;  // compilation error!
        B b = 1;
        return 0;
}
```

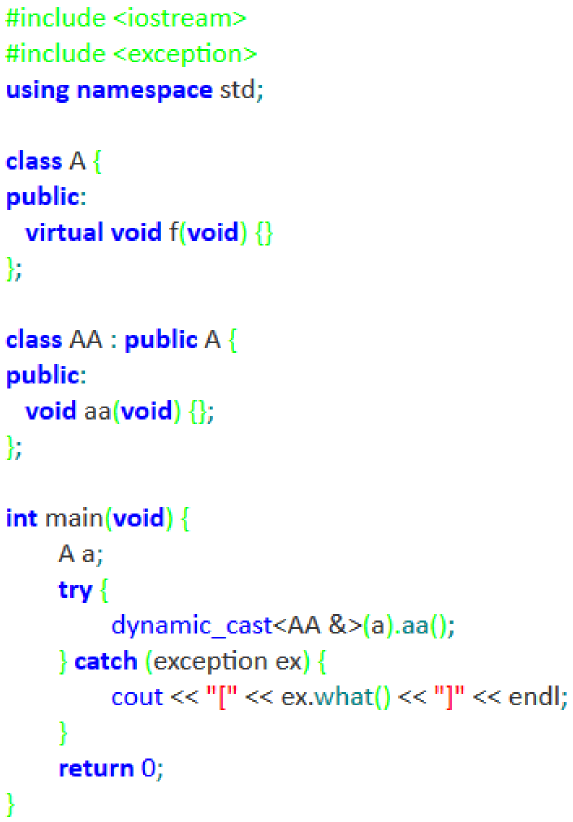
The ‘explicit’ keyword

- Note that the following function would be wrong, too:
  - A fun(**void**) { **return** 0; }
- while this one wouldn’t:
  - B fun(**void**) { **return** 0; }

The ‘exception’ class

- The *exception* class is a **base** (or a **root**) for all other predefined exceptions.
- It contains a function called *what*, which is designed to provide a pointer to the so-called “C”-style string (a character sequence terminated with a *null* character) **describing the nature of the exception**.

The ‘exception’ class



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

class A {
public:
  virtual void f(void) {}
};

class AA : public A {
public:
  void aa(void) {};
};

int main(void) {
        A a;
        try {
                dynamic_cast<AA &>(a).aa();
        } catch (exception ex) {
                cout << "[" << ex.what() << "]" << endl;
        }
        return 0;
}
```

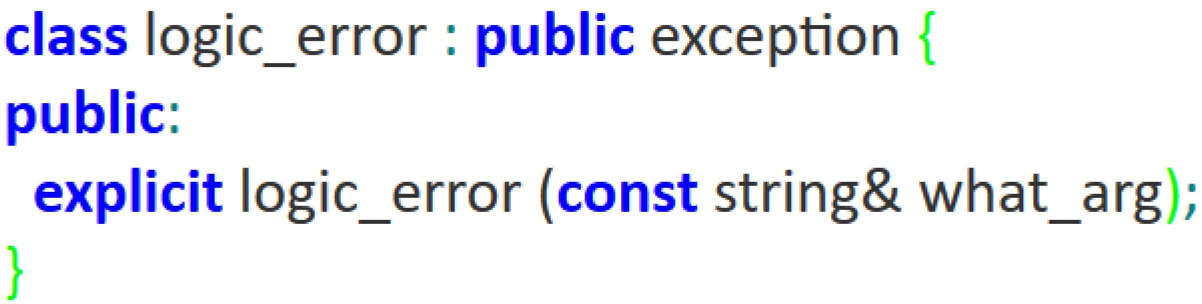
- We will get “[Bad dynamic_cast!]

The ‘logic_error’ class

- exception ← logic_error
- The *logic_error* class is directly derived from the *exception* class.
- It’s designed to represent all the exceptions caused by **a violation of the rules imposed by the logic of the algorithm/program**.
- It may (but doesn’t always) mean that exceptions of this kind are **preventable**, i.e. they won’t happen if all the processed data is vali

The ‘logic_error’ class

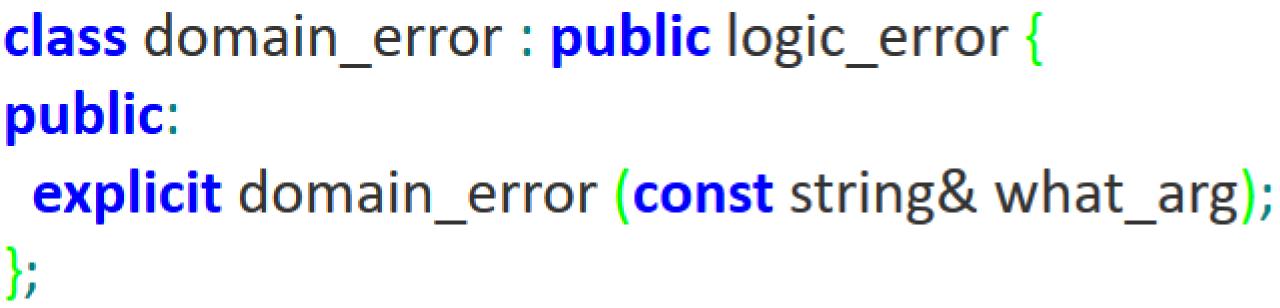
- The constructor of the class allows us to “pack” a detailed message inside the exception object.
- The following directive is **mandatory** in a code that makes use of these classes:
  - #include <stdexcept>



```
class logic_error : public exception {
public:
  explicit logic_error (const string& what_arg);
}
```

The ‘domain_error’ class

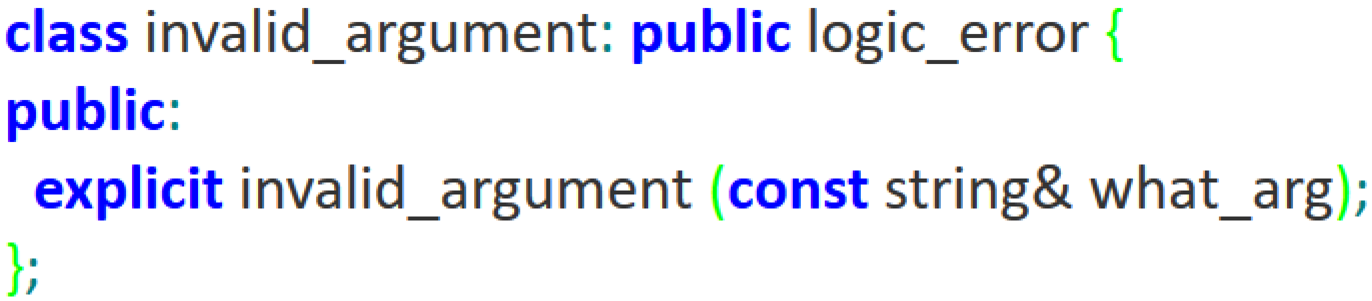
- exception ← logic_error ← domain_error
- The *domain_error* class is derived from the *logic_error* class. It’s designed to represent all exceptions caused by the **data exceeding the permissible range**.



```
class domain_error : public logic_error {
public:
  explicit domain_error (const string& what_arg);
};
```

The ‘invalid_argument’ class

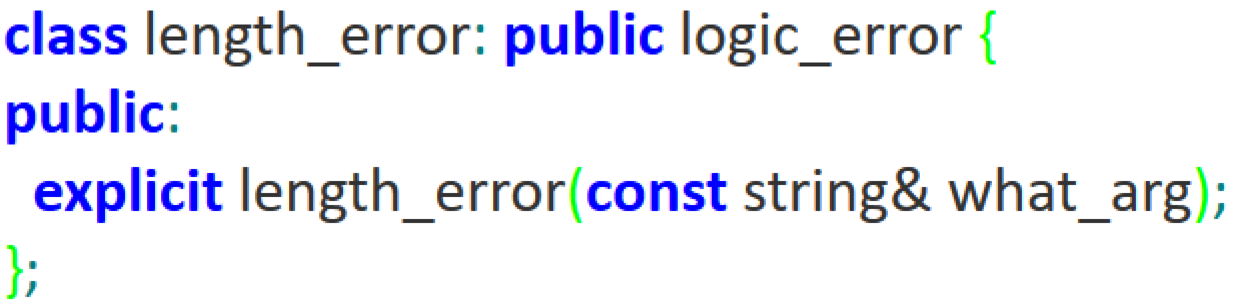
- exception ← logic_error ← invalid_argument
- The *invalid_argument* class is derived from the *logic_error* class. It’s designed to represent all exceptions caused by **passing improper arguments to methods or functions or constructors**.



```
class invalid_argument: public logic_error {
public:
  explicit invalid_argument (const string& what_arg);
};
```

The ‘length_error’ class

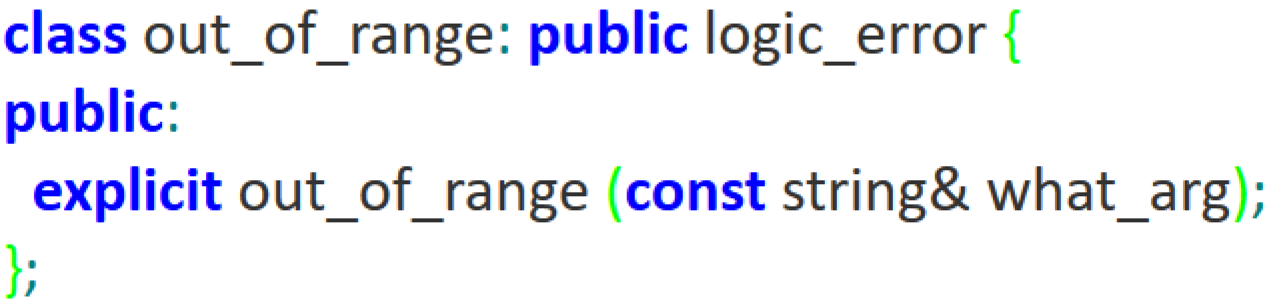
- exception ← logic_error ← length_error
- The *length_error* class is derived from the *logic_error* class. It’s designed to represent all exceptions caused by **using illegal values to specify size/length of data aggregates**.



```
class length_error: public logic_error {
public:
  explicit length_error(const string& what_arg);
};
```

The ‘out_of_range’ class

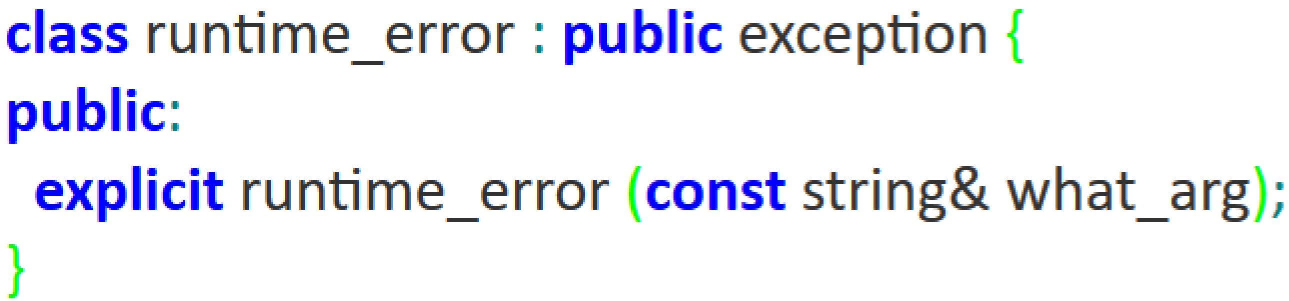
- exception ← logic_error ← out_of_range
- The *out_of_range* class is derived from the *logic_error* class. It’s designed to represent exceptions connected to the **use of illegal indexes/keys while accessing numbered/keyed data collections**.



```
class out_of_range: public logic_error {
public:
  explicit out_of_range (const string& what_arg);
};
```

The ‘runtime_error’ class

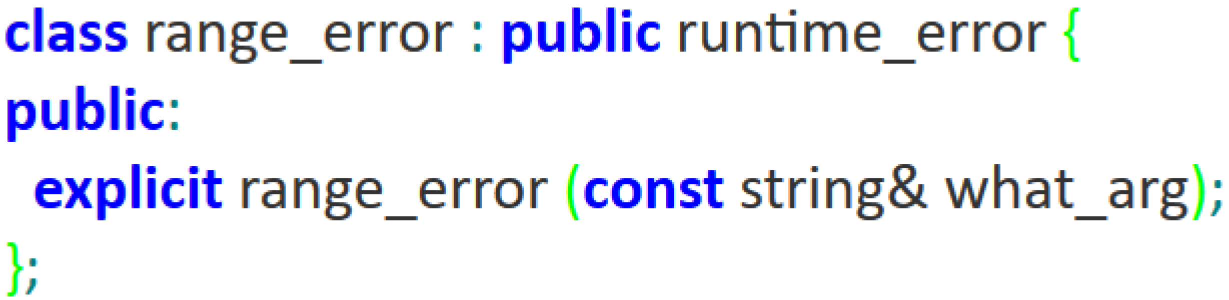
- exception ← runtime_error
- The *runtime_error* class is derived directly from the *exception* class. It’s designed to represent all exceptions caused by **circumstances which may occur during the execution of the program**.



```
class runtime_error : public exception {
public:
  explicit runtime_error (const string& what_arg);
}
```

The ‘range_error’ class

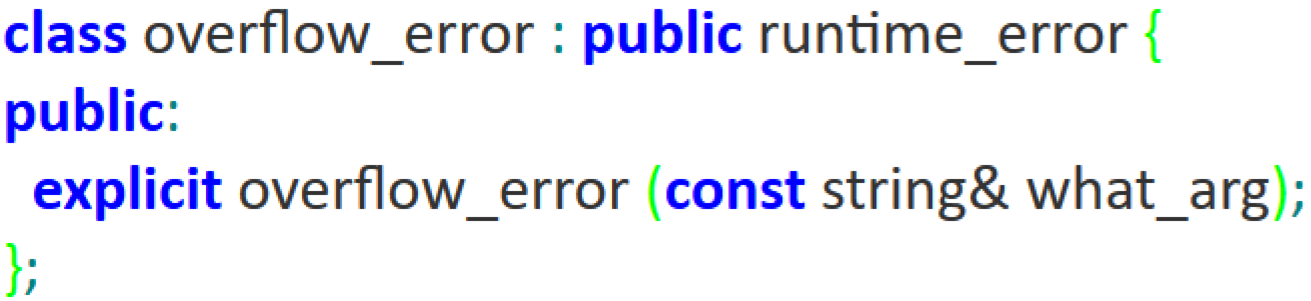
- exception ← runtime_error ← range_error
- The *range_error* class is derived from the *runtime_error* class. It’s designed to represent exceptions caused by **obtaining computation results exceeding the permissible range**.



```
class range_error : public runtime_error {
public:
  explicit range_error (const string& what_arg);
};
```

The ‘overflow_error’ class

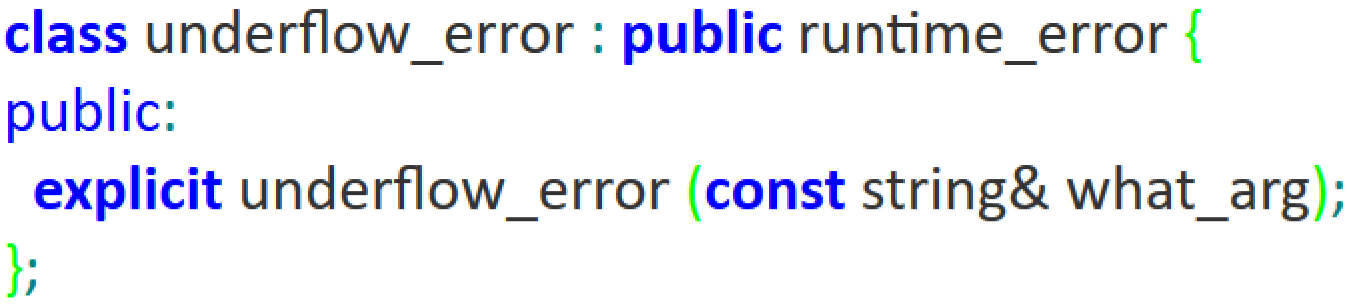
- exception ← runtime_error ← overflow_error
- The *overflow_error* class is derived from the *runtime_error* class. It’s designed to represent exceptions caused by **obtaining results too large to represent any useful value** (in the domain sense).



```
class overflow_error : public runtime_error {
public:
  explicit overflow_error (const string& what_arg);
};
```

The ‘underflow_error’ class

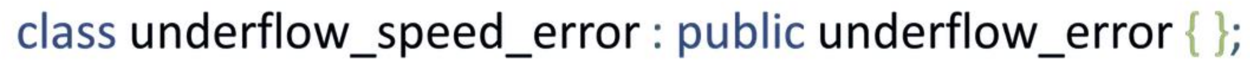
- exception ← runtime_error ← underflow_error
- The *underflow_error* class is derived from the *runtime_error* class. It’s designed to represent exceptions caused by **obtaining results too small to represent any useful value** (in the domain sense).



```
class underflow_error : public runtime_error {
public:
  explicit underflow_error (const string& what_arg);
};
```

What next?

- if you want to create a specialized category of exceptions designed to distinguish a very specific class of underflow errors, you can do it in this way:



```
class underflow_speed_error : public underflow_error { };
```

bad_alloc

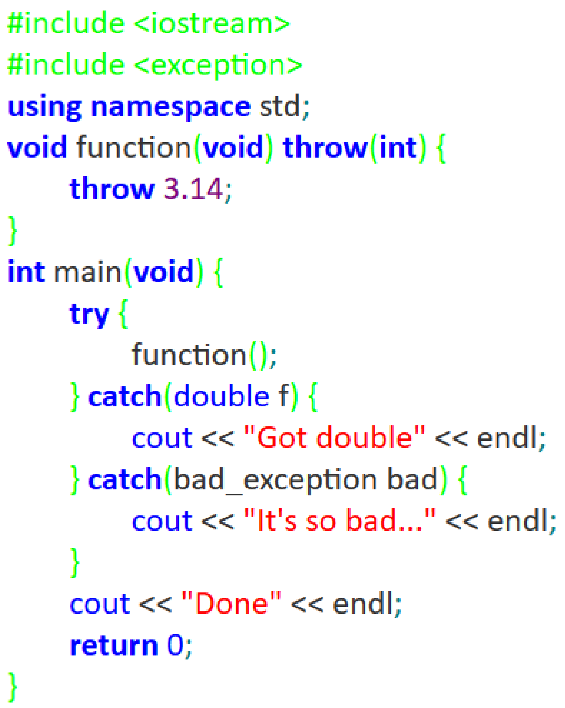
- The *bad_alloc* exception may be thrown as an undesired effect of invoking the *new* or *new[]* operators when the runtime or operating system can’t fulfil our memory requirements.

exception ← bad_alloc

bad_exception

- exception ← bad_exception
- The *bad_exception* exception is thrown when a function tries to throw an exception not specified inside its throw specification.
- Note that this exception cannot be caught directly.

bad_exception



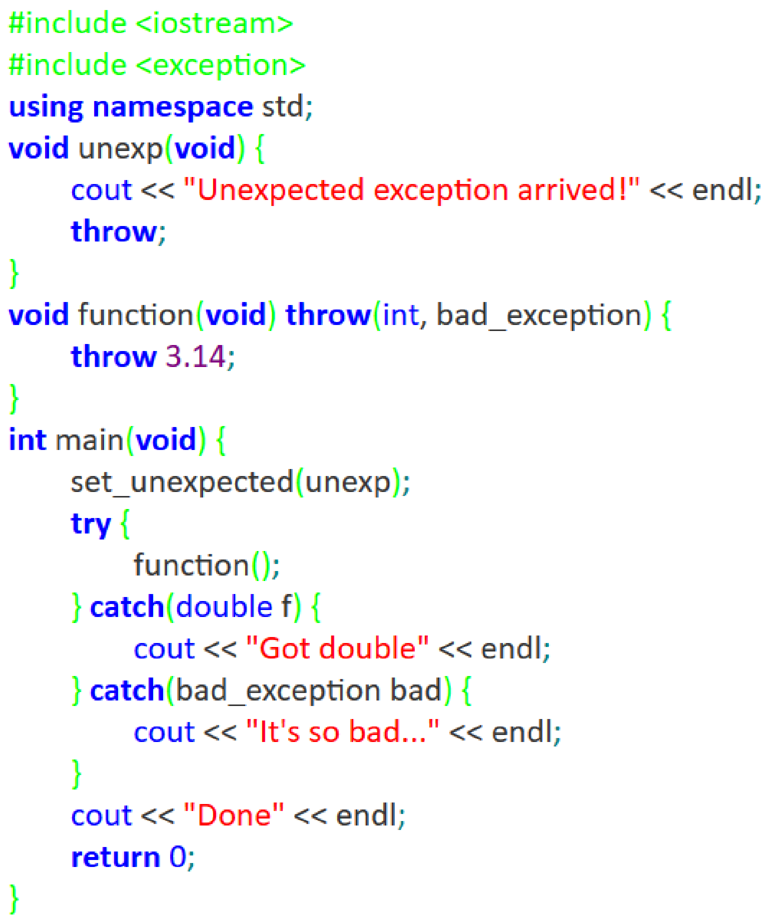
```
#include <iostream>
#include <exception>
using namespace std;
void function(void) throw(int) {
      throw 3.14;
}
int main(void) {
      try {
            function();
      } catch(double f) {
            cout << "Got double" << endl;
      } catch(bad_exception bad) {
            cout << "It's so bad..." << endl;
      }
      cout << "Done" << endl;
      return 0;
}
```

- The program **doesn’t output either “It's so bad...” or “Done”, or even “Got double” messages.**

bad_exception

- Proper handling of the *bad_exception* exception requires the function to **specify *bad_exception* on its throw list (it looks like a paradox but it's true), and the unexpected handler function must be defined and set**.
- Failure to meet any of these conditions will result in undesired program behaviour.

bad_exception



```
#include <iostream>
#include <exception>
using namespace std;
void unexp(void) {
	cout << "Unexpected exception arrived!" << endl;
	throw;
}
void function(void) throw(int, bad_exception) {
	throw 3.14;
}
int main(void) {
	set_unexpected(unexp);
	try {
		function();
	} catch(double f) {
		cout << "Got double" << endl;
	} catch(bad_exception bad) {
		cout << "It's so bad..." << endl;
	}
	cout << "Done" << endl;
	return 0;
}
```

bad_exception

- The program will output the following lines to the screen:
  - Unexpected exception arrived!
  - It's so bad...
  - Done