



# Laboratorio di Tecnologie dell'Informazione

Ing. Marco Bertini  
marco.bertini@unifi.it  
<http://www.micc.unifi.it/bertini/>



# Exceptions



# What are exceptions ?

- Exceptions are a mechanism for handling an error during execution.
- A function can indicate that an error has occurred by throwing an exception.
- The code that deals with the exception is said to handle it.



# Why use exceptions ?

- Code where the error occurs and code to deal with the error can be separated
- Exceptions can be used with constructors and other functions/operators which can not return an error code
- Properly implemented exceptions lead to better code



# How to use exceptions ?

- **try**
  - Try executing some block of code
  - See if an error occurs
- **throw**
  - An error condition occurred
  - Throw an exception to report the failure
- **catch**
  - Handle an exception thrown in a try block



# How exceptions work ?

- Normal program control flow is halted
  - At the point where an exception is thrown
- The program call stack “unwinds”
  - Stack frame of each function in call chain “pops”
  - Variables in each popped frame are destroyed
  - Goes until an enclosing try/catch scope is reached
- Control passes to first matching catch block
  - Can handle the exception and continue from there
  - Can free some resources and re-throw exception



# What's right about exceptions

- Can't be silently ignored: if there is no applicable catch block for an exception the program terminates
- Automatically propagate across scopes (due to stack unwinding)
- Handling is out of main control flow, the code that implements the algorithm is not polluted



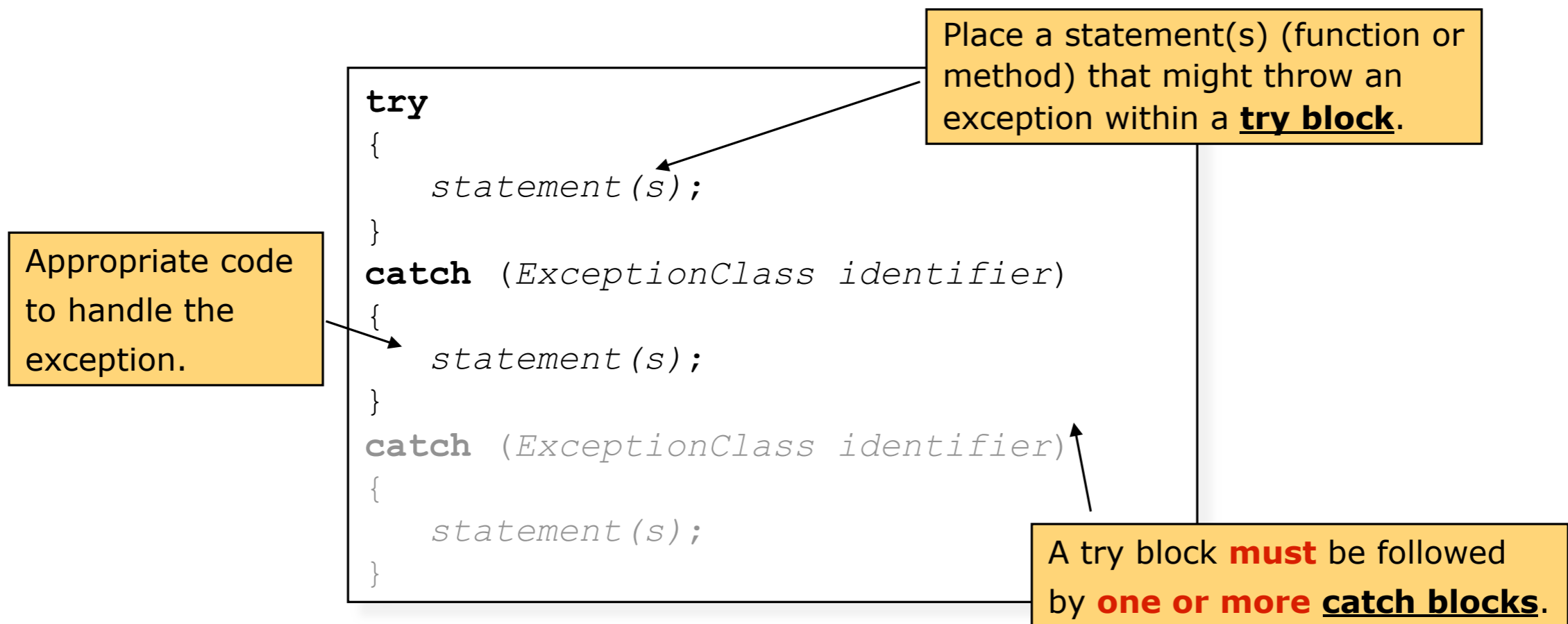
# Exceptions syntax





# C++ exceptions syntax

- Use try-catch blocks to catch an exception





# C++ exception flow

- When a statement (function or method) in a try block causes an exception:
  - Rest of try block is ignored.
  - Control passes to catch block corresponding to the exception.
  - After a catch block executes, control passes to statement after last catch block associated with the try block.

```
try
{
    ...
    statement;
    ...
}
catch (ExceptionClass identifier)
{
    statement(s);
}
statement(s);
```

Throw an exception



# C++ exception flow - cont.

- A more complex example of exception flow:

```
void encodeChar(int i, string& str)
{
    ...
    str.replace(i, 1, 1, newChar);
}
```

Can throw the *out\_of\_range* exception.

```
void encodeString(int numChar, string& str)
{
    for(int i=numChar-1; i>=0; i--)
        encodeChar(i, str);
}
```

```
int main()
{
    string str1 = "NTU IM";
    encodeString(99, str1);
    return 0;
}
```

Abnormal program termination



# Catching the exception

- Two examples on how to catch the exception:

```
void encodeChar(int i, string& str)
{
    try
    {
        ...
        str.replace(i, 1, 1, newChar);
    } catch (out_of_range e) {
        cout << "No character at " << i << endl;
    }
}
```

```
void encodeString(int numChar, string& str)
{
    for(int i=numChar-1; i>=0; i--)
        encodeChar(i, str);
}
```

```
int main()
{
    string str1 = "NTU IM";
    encodeString(99, str1);
    return 0;
}
```

No character at 98  
No character at 97  
...



# Catching the exception

- Two examples on how to catch the exception:

```
void encodeChar(int i, string& str)
{
    ...
    str.replace(i, 1, 1, newChar);
}
```

```
void encodeString(int numChar, string& str)
{
    try
    {
        for(int i=numChar-1; i>=0; i--)
            encodeChar(i, str);
    } catch (out_of_range e) {
        cout << "Something wrong" << endl;
    }
}
```

```
int main()
{
    string str1 = "NTU IM";
    encodeString(99, str1);
    return 0;
}
```

Something wrong



# Handlers

- A handler may re-throw the exception that was passed:
    - it forwards the exception
    - Use: `throw; //` no operand
    - after the local handler cleanup it will exit the current handler
  - A handler may throw an exception of a different type
    - it translates the exception
-



# Catching multiple exceptions

- The order of catch clauses is important:
    - Especially with inheritance-related exception classes
    - Put more specific catch blocks before more general ones
    - Put catch blocks for more derived exception classes before catch blocks for their respective base classes
  - `catch(...)` catches any type
-



# Catching multiple exceptions example

```
try {  
    // can throw exceptions  
} catch (DerivedExc &d) {  
    // Do something  
} catch (BaseExc &d) {  
    // Do something else  
} catch (...) {  
    // Catch everything else  
}
```

---





# Throwing exceptions

- When you detect an error within a method, you can throw an exception by using a throw statement.
- The remaining code within the function does not execute.
- Syntax: `throw ExceptionClass(stringArgument);`

type of the exception

more detailed information

```
void myMethod(int x) throw(MyException)
{
    if (...)
        throw MyException("MyException: ...");
    ...
} // end myMethod
```



# Throwing exceptions - cont.

- The exception is propagated back to the point where the function was called.

```
try
{
    ...
    myMethod(int x);
    ...
}
catch (ExceptionClass identifier)
{
    statement(s);
}
```

The diagram illustrates the flow of an exception. A yellow box labeled "back to here!!" has an arrow pointing to the `myMethod(int x);` line in the `try` block. A red arrow then curves from the end of the `myMethod(int x);` line down to the `catch (ExceptionClass identifier)` block, indicating that the exception is caught at this point.



# Specifying exceptions

- Functions that throw an exception have a throw clause, to restrict the exceptions that a function can throw.
  - Allow stronger type checking enforced by the compiler
  - By default, a function can throw anything it wants
- A throw clause in a function's signature
  - Limits what can be thrown
  - A promise to calling function
- A throw clause with no types
  - Says nothing will be thrown
- Can list multiple types, comma separated



# Specifying exceptions examples

```
// can throw anything  
void Foo::bar();
```

```
// promises not to throw  
void Foo::bar() throw();
```

```
// promises to only throw int  
void Foo::bar() throw(int);
```

```
// throws only char or int  
void Foo::bar() throw(char,int);
```

---



# Destructors and exceptions



# Destructors and exceptions

- Prevent exceptions from leaving destructors: premature program termination or undefined behaviour can result from destructors emitting exceptions
- during the stack unwinding resulting from the processing of the exception are called the destructors of local objects, and one may trigger another exception



# How to behave: example

```
class DBConnection {
public:
    //...

    // return a DBConnection object
    static DBConnection create();

    void close(); // close connection and
                 // throws exception if
                 // closing fails
};
```

```
// class to manage DBConnection
class DBConnMgr {
public:
    //...
    DBConnMgr(DBConnection dbc);
    ~DBConnMgr() {
        dbc.close(); // we're sure it
                    // gets closed
    }

private:
    DBConnection db;
};
```

```
// client code
{
    DBConnMgr dbc( DBConnection::create() );
    //... use DBConnection through DBConnMgr interface
} // DBConnMgr obj is automatically destroyed, calling the close
```



# How to behave: example

```
class DBConnection {  
public:  
    //...  
  
    // return a DBConnection object  
    static DBConnection create();  
  
    void close(); // close connection and  
                // throws exception if  
                // closing fails  
};
```

```
// class to manage DBConnection  
class DBConnMgr {  
public:  
    //...  
    DBConnMgr(DBConnection dbc);  
    ~DBConnMgr() {  
        db.close(); // we're sure it  
                   // gets closed  
    }  
};
```

**If close() throws the destructor propagates the exception**

```
// client code  
{  
    DBConnMgr dbc( DBConnection::create() );  
    //... use DBConnection through DBConnMgr interface  
} // DBConnMgr obj is automatically destroyed, calling the close
```





# (Not so good) solutions

- Terminate the program:

```
DBConnMgr::~~DBConnMgr() {  
    try{ db.close(); }  
    catch (...) {  
        // log failure and...  
        std::abort();  
    }  
}
```

- Swallow the exception:

```
DBConnMgr::~~DBConnMgr() {  
    try{ db.close(); }  
    catch (...) {  
        // just log the error  
    }  
}
```



# (Not so good) solutions

- Terminate the program:

```
DBConnMgr::~~DBConnMgr() {  
    try{ db.close(); }  
    catch (...) {  
        // log failure and...  
        std::abort();  
    }  
}
```

- Swallow the exception:

```
DBConnMgr::~~DBConnMgr() {  
    try{ db.close() }  
    catch (...) {  
        // just log the error  
    }  
}
```

With this solution we're just hiding the problem



# A better strategy

```
// class to manage DBConnection
class DBConnMgr {
public:
    //...
    DBConnMgr(DBConnection dbc);
    void close() {
        dbc.close();
        closed = true;
    }
    ~DBConnMgr() { // we're sure it gets closed
        if( !closed ) {
            try {
                dbc.close();
            } catch (...) {
                // log and... terminate or swallow
            }
        }
    }
private:
    DBConnection dbc;
    bool closed;
};
```



# A better strategy

```
// class to manage DBConnection
class DBConnMgr {
public:
    //...
    DBConnMgr(DBConnection dbc);
    void close() {
        dbc.close();
        closed = true;
    }
    ~DBConnMgr() { // we're sure it gets closed
        if( !closed ) {
            try {
                dbc.close();
            } catch (...) {
                // log and... terminate or swallow
            }
        }
    }
private:
    DBConnection db;
    bool closed;
};
```

Client code should use  
this method...





# A better strategy

```
// class to manage DBConnection
class DBConnMgr {
public:
    //...
    DBConnMgr(DBConnection dbc);
    void close() {
        dbc.close();
        closed = true;
    }
    ~DBConnMgr() { // we're sure it gets closed
        if( !closed ) {
            try {
                dbc.close();
            } catch (...) {
                // log and... terminate or swallow
            }
        }
    }
private:
    DBConnection db;
    bool closed;
};
```

Client code should use this method...

...but if it doesn't there's the destructor



# Defining exceptions classes

Syntax and example



# Defining exceptions classes

- C++ Standard Library supplies a number of exception classes.
- E.g., `exception`, `out_of_range`, ... etc.
- You may also want to define your own exception class.
- Should inherit from those pre-defined exception classes for a standardized exception working interface.
- Syntax:  

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



# Defining exceptions classes example

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

class MyException : public exception
{
public:
    MyException(const string & Message = "")
        : exception(Message.c_str()) {}
} // end class
```

```
try
{
    ...
}
catch (MyExceptoin e)
{    cout << e.what();
}
```

```
throw MyException("more detailed information");
```





# A full example

- An ADT List implementation using exceptions:
  - out-of-bound list index.
  - attempt to insert into a full list.



# Define two exception classes

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

class ListIndexOutOfRangeException : public out_of_range {
public:
    ListIndexOutOfRangeException(const string& message = "")
        : out_of_range(message.c_str()) {}
}; // end ListException

class ListException : public logic_error {
public:
    ListException(const string & message = "")
        : logic_error(message.c_str()) {}
}; // end ListException
```



# Declare the throw

```
#include "MyListExceptions.h"
. . .
class List
{
public:
. . .
void insert(int index, const ListItemType& newItem)
    throw(ListIndexOutOfRangeException,
          ListException);
. . .
} // end List
```



# Method implementation

```
void List::insert(int index, const ListItemType& newItem)
    throw(ListIndexOutOfRangeException, ListException) {
    if (size >= MAX_LIST)
        throw ListException("ListException: List full on insert");
    if (index >= 1 && index <= size+1) {
        for (int pos = size; pos >= index; --pos)
            items[translate(pos+1)] = items[translate(pos)];
        // insert new item
        items[translate(index)] = newItem;
        ++size; // increase the size of the list by one
    } else // index out of range
        throw ListIndexOutOfRangeException(
            "ListIndexOutOfRangeException: Bad index on insert");
} // end insert
```



# Good Programming Style with C++ Exceptions

- Don't use exceptions for normal program flow
  - Only use where normal flow isn't possible
- Don't let exceptions leave main or constructors
  - Violates "normal" initialization and termination
- Always throw some type
  - So the exception can be caught
- Use exception specifications widely
  - Helps caller know possible exceptions to catch



# Exception-safe functions

- Exception-safe functions offer one of three guarantees:
- **basic guarantee:** if an exception is thrown, everything in the program remains in a valid state
- **strong guarantee:** if an exception is thrown, the state of the program is unchanged. The call to the function is atomic
- **nothrow guarantee:** promise to never throw exception: they always do what they promise. All operations on built-in types are nothrow.



# Exception-safe code

- When an exception is thrown, exception safe functions:
  - leak no resource (e.g. new-ed objects, handles, etc.)
  - don't allow data structures to become corrupted (e.g. a pointer that had to point to a new object was left pointing to nowhere)



# Credits

- These slides are (heavily) based on the material of:
  - Dr. Walter E. Brown, Fermi Lab
  - Dr. Chien Chin Chen, National Taiwan University
  - Dr. Jochen Lang, University of Ottawa
  - Fred Kuhns, Washington University
  - Scott Meyers, “Effective C++, 3rd ed.”