CSE 165/ENGR 140
Intro to Object Orient Program
Lecture 21 – Exception Handling
Announcement

- Lab #8
  - Due 4/16 at 11:59PM

- Quiz: 4/16

- Final project (in UCMCROPS)
  - Due date: 5/11 (Monday) at 11:59PM
  - Presentation date: 5/12 (Tuesday) at 3:00PM

- Reading assignment:
  - Ch. 9 (Vol. 2)
Exception Handling

- Special mechanism available in C++ using the `try` and `catch` keywords:

```cpp
// exceptions example
#include <iostream>
using namespace std;

int main () {
    try {
        if ( problem_detected() )
            throw 20;
    }
    catch (int e) {
        cout << "An exception occurred. Exception Number: " << e << endl;
    }
    return 0;
}
```
Many “catches” can be used to selectively deal with different errors:

```java
try {
    // Code that may generate exceptions
} catch (type1 id1) {
    // Handle exceptions of type1
} catch (type2 id2) {
    // Handle exceptions of type2
} catch (type3 id3)
    // Etc...
} catch (typeN idN)
    // Handle exceptions of typeN
}
// Normal execution resumes here...
```
Exception Handling

- You may throw exceptions of your own types:

```cpp
//: C01:MyError.cpp
class MyError {
    const char* const data;
public:
    MyError(const char* const msg = 0) : data (msg) {}
    void print() {cout << data << endl;}
};

void f ( int i ) {
    if ( i<0 ) // Here we "throw" an exception object:
        throw MyError("something bad happened");
}

int main() {
    try {
        f ( read_integer_input() );
    }
    catch ( MyError e )
    {
        e.print();
    }
}
When an exception is thrown:
- the exception-handling system looks through the “nearest” handlers in the order they appear in the source code.
- When it finds a match, the exception is considered handled and no further searching occurs.

But...
- you may decide to re-throw the same exception again so that other parts of the code can also handle it.
- This is useful, for example, when your code has to perform critical operations in response to an exception, and you also want users of your code to know that the exception happened.
Exception Handling

- Catching any exception
  - You can create a handler that catches *any* type of exception (catchall).
  - When an exception is caught, resource will be released and the exception will be re-thrown to allow other handlers to process it.

```cpp
    catch(...) {
        cout << "an exception was thrown" << endl;
        // Deallocate your resource here, and then re-throw...
        throw;
    }
```
If no one catches an exception...

- The special library function `terminate()` (declared in the `<exception>` header) is automatically called.
- By default, `terminate()` calls the Standard C library function `abort()`, which abruptly exits the program.
- Destructors for global and static objects do not execute.
- You can install your own `terminate()` function using the `set_terminate()` function.
//: C01:Terminator.cpp - Use of set_terminate()
#include <exception>
#include <iostream>
#include <cstdlib>
using namespace std;

void terminator() {
    cout << "I'll be back!" << endl;
    exit(0);
}

void (*old_terminate)() = set_terminate(terminator);

class A {
public:
    class B {};
    void f() { cout << "A::f()" << endl;
        throw B();
    }
    ~A() { throw 'c'; }
};

int main() {
    try { A a; a.f();
        }
catch(...) {
        cout << "inside catch(...)" << endl;
    }
}
Standard Exceptions

- All standard exception classes derive from the class `exception`, defined in the header `<exception>`
  - The two main derived classes are:
    - `logic_error`
      - Errors such as passing an invalid argument.
    - `runtime_error`
      - Errors related to resources, such as hardware failure or memory exhaustion.
      - They provide a constructor that takes a `std::string` argument so that you can store a message in the exception object and extract it later with `exception::what()`
    - Found in `<stdexcept>` (which itself includes `<exception>`)
Standard Exceptions

//: C01:StdExcept.cpp
// Derives an exception class from std::runtime_error
#include <stdexcept>
#include <iostream>
using namespace std;

class MyError : public runtime_error {
public:
   MyError(const string& msg = "") : runtime_error(msg) {};
int main() {
   try {
      throw MyError("my message");
   }
   catch (MyError& x) {
      cout << x.what() << endl;
   }
}
## Exception classes derived from logic_error

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain_error</td>
<td>Reports violations of a precondition.</td>
</tr>
<tr>
<td>invalid_argument</td>
<td>Indicates an invalid argument to the function from which it’s thrown.</td>
</tr>
<tr>
<td>length_error</td>
<td>Indicates an attempt to produce an object whose length is greater than or</td>
</tr>
<tr>
<td></td>
<td>equal to npos (the largest representable value of type size_t).</td>
</tr>
<tr>
<td>Out_of_range</td>
<td>Reports an out-of-range argument.</td>
</tr>
<tr>
<td>Bad_cast</td>
<td>Thrown for executing an invalid dynamic_cast expression in runtime type</td>
</tr>
<tr>
<td></td>
<td>identification (see Chapter 8).</td>
</tr>
<tr>
<td>bad_typeid</td>
<td>Reports a null pointer p in an expression typeid(p). (Again, a runtime</td>
</tr>
<tr>
<td></td>
<td>type identification feature in Chapter 8).</td>
</tr>
</tbody>
</table>
## Classes derived from runtime_error

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>range_error</td>
<td>Reports violation of a postcondition.</td>
</tr>
<tr>
<td>overflow_error</td>
<td>Reports an arithmetic overflow.</td>
</tr>
<tr>
<td>bad_alloc</td>
<td>Reports a failure to allocate storage.</td>
</tr>
</tbody>
</table>
Some compilers allow you to choose whether or not to support exception handling.

- Code that supports exception handling can be much larger:
  - to keep track of destructors that may need to be called if functions throw exceptions, for example.
- Robust error handling is essential, so either all the functions you use return possible errors, or the exception handling mechanism is used.