How To Report Errors?

- **C style of error reporting:** return values
  - For C Standard Library/UNIX functions, 0 usually means success, < 0 means error
  - Same for Windows API (**HRESULT** data type)
  - Additional error details must be stored elsewhere

- **Not very informative!**
  - Not all errors have the same details

- **Propagating internal errors is also not clean**
  - Enclosing function must explicitly pass along error
  - May accidentally mangle useful error information
C++ Exception Handling

- A mechanism for handling errors at runtime
- Code that can detect a problem, but might not know how to handle it, can *throw* an exception.
  - The “exception” is a value describing what happened.
- Code that knows how to handle the problem *catches* the exception.
  - Could be the immediate caller of the function, but it doesn’t have to be.
  - Caught exception specifies what happened to cause the problem.
- Complementary to other error-handling approaches, e.g. assertions.
Throwing Exceptions

- An exception is a value describing the error
  - Could be any object, or even a primitive!
  - Usually a specific class (or set of classes) is used for representing errors
    - C++ Standard Library provides exception classes
    - People often create their own, too
  - Exception’s type usually indicates the general kind of error

Example:
```cpp
double computeValue(double x) {
    if (x < 3.0)
        throw invalid_argument("x must be >= 3.0");

    return 0.5 * sqrt(x - 3.0);
}
```
Catching Exceptions

- To catch an exception, use a try-catch block

```cpp
double x;
cout << "Enter value for x: ";
cin >> x;
try {
    double result = computeValue(x);
    cout << "Result is " << result << endl;
} catch (invalid_argument) {
    cout << "An invalid value was entered." << endl;
}
```

- Code within `try` block *might* throw exceptions
- Specify what kind of exception can be handled at start of `catch` block
Using Caught Exceptions

- Can name the caught exception:
  ```cpp
double x;
cout << "Enter value for x: ";
cin >> x;
try {
    double result = computeValue(x);
    cout << "Result is " << result << endl;
}
catch (invalid_argument &e) {
    cout << "Error: " << e.what() << endl;
}
```

- This is better – can pass details of error in the exception
- All C++ standard exception classes have `what()` function
More Exception Details

```cpp
try {
    double result = computeValue(x);
    cout << "Result is " << result << endl;
}
catch (invalid_argument &e) {
    cout << "Error: " << e.what() << endl;
}
```

- If `computeValue()` throws, execution transfers *immediately* to `catch` handler
  - “Result is…” operation is skipped.
- Usual variable scoping rules apply to `try` and `catch` blocks
  - Scope of `result` is only within `try` block
    - `catch`-block cannot refer to variables declared within `try`-block
  - Scope of `e` is only within `catch` block
Catching Multiple Exceptions

- Can specify multiple catch blocks after a single try block
  
  ```
  try {
    performTask(config);
  }
  catch (invalid_argument &ia) { // Invalid args.
    cout << ia.what() << endl;
  }
  catch (bad_alloc &ba) { // Out of memory.
    cout << "Ran out of memory." << endl;
  }
  catch (exception &e) { // Something else???
    cout << "Caught another standard exception: " << e.what() << endl;
  }
  ```

- Order matters! *First* matching catch-block is chosen.
- *Only one* catch block is executed.
Handlers that Throw

- **catch** blocks can also throw exceptions, if necessary
  ```
  try {
    runOperation();
  }
  catch (ProcessingError &pe) {
    if (!recover())
      throw FatalError("Couldn't recover!");
  }
  catch (FatalError &fe) {
    // Called when runOperation() throws, but
    // not when a previous catch block throws.
    cerr << "Couldn't compute!" << endl;
  }
  ```

- If **ProcessingError** catch-block throws another exception, it propagates out of entire try/catch block
  - Only exceptions thrown within the **try** block are handled
Catching Everything…

- To catch *everything*, use ... for the exception type
  ```
  try {
      doSomethingRisky();
  }
  catch (...) {
      // Catches ANY kind of exception
      cout << "Hmm, caught something..." << endl;
  }
  ```

- Problem: no type information about the exception!
  - Limits its general usefulness

- Usually used when code needs to guarantee that no exceptions can escape from the *try*-block

- Also used in test code, for verifying exception-throwing behavior of functions
Obj3D * findObject(int objID) {
    Obj3D *pObj = ...      // Find the object
    if (pObj == 0)
        throw BadObjectID(objID);
    return pObj;
}

void rotateObject(int objID, float angle) {
    Matrix m;
    // This could throw, but let the exception propagate
    Obj3D *pObj = findObject(objID);
    m.setRotateX(angle);
    pObj->transform(m);
}

- If `findObject()` throws an exception:
  - The function stops executing where the exception is thrown
  - Its local variables are cleaned up automatically
Obj3D * findObject(int objID) {  
    Obj3D *pObj = ...  // Find the object  
    if (pObj == 0)  
        throw BadObjectID(objID);  
    return pObj;  
}  

void rotateObject(int objID, float angle) {  
    Matrix m;  // This could throw, but let the exception propagate  
    Obj3D *pObj = findObject(objID);  
    m.setRotateX(angle);  
    pObj->transform(m);  
}

- rotateObject() doesn’t try to catch any exceptions!
  - If findObject() throws, it will propagate out of rotateObject()  
  - rotateObject() will also stop executing where exception occurs
Exceptions and Functions

- If an exception is not caught within a function, that function terminates.
  - Local variables go out of scope…
  - They are cleaned up via destructor calls, as usual

- If the calling function doesn’t handle the exception, it also terminates.
  - …and so forth, all the way up the stack…
  - This is called “stack unwinding”

- If main() doesn’t handle the exception, the entire program terminates.
Catching Exceptions

- **Catch a reference to the exception**
  - Exceptions are passed *by-value* if `catch`-block doesn’t use a reference
  ```cpp
  try {
    riskyBusiness();
  }
  catch (MyException e) {
    ...
  }
  ```
  - The exception is **copied** into `e` when it is caught
  - Using "*MyException &e*" passes the exception by reference

- **Several other good reasons to catch by reference!**
  - See *More Effective C++*, Item 13, for even more details…
Arrays of objects:

```
MyClass *array = new MyClass[100];
```

- Individual objects are initialized using default constructor
- `delete[] array;`
- `MyClass` destructor is called on each object, then array memory is reclaimed

What happens if one of the destructor calls throws an exception?

- Entire clean-up process is derailed! Memory leaks!

Class destructors should *never* throw exceptions.

- If any code in destructor *could* throw, wrap with `try/catch` block. Don’t let it propagate out of the destructor!
- (C++ standard says `delete` and `delete[]` won’t throw.)
Other Exception Tips

- Exceptions are usually *not* caught by the same function that they are thrown in
  - Exceptions are usually used to signal *to the function’s caller* when an operation fails

- If you have a function that throws exceptions to itself
  - The function may be too large and should probably be broken into several functions
  - Or, exceptions may not be appropriate for what you are doing!

- Exception handling is a complex subject
  - Only scratched the surface here…
  - Take Advanced C++ track, and/or get some good books!
Exceptions vs. Assertions

- When to use exceptions instead of assertions?
  - ...and when assertions instead of exceptions?
- Assertions are usually only available during your software development
  - Usually compiled out of a program or library before it is given to customers/peers/etc.
  - Diagnosing an assertion failure requires access to the source code – not always possible, or even desirable!
- Exceptions are part of a program’s public interface
  - They are not compiled out of a program before others use your code
  - Other people developing against your code may want or need to handle your exceptions
Use assertions when:
- Checking for your own programming bugs
- Checking arguments for internal/private functions
- No third party will ever call your API or want to know about the errors being detected

Use exceptions when:
- You are releasing a public API for others to use
- Third parties may want/need to handle your API’s errors

Always document what exceptions are thrown, and the situations that cause them to be thrown!
- Just as important as documenting what functions do, and what their arguments are for!
Generic Programming

- A lot of concepts are independent of data type
  - Example: your SparseVector class
  - How to use it with float, double, … data?
  - Or, another class data type, like complex?

- One “solution” – copy the code.
  - FloatSparseVector, ComplexSparseVector, …
  - A maintenance nightmare!

- The C solution: use untyped pointers – void*
  - Can point to a value of any type, but all type-checking disappears!
C++ Templates

- C++ introduces templates
  - Allows a class or function to be parameterized on types and constants

- A template isn’t itself a class or function…
  - More like a pattern for classes or functions

- To use a template, you instantiate it by supplying the template parameters
  - The result is a class or a function
  - “Generating a class/function from the template.”
A Simple Template Example

- **Our Point class**, which uses **double** values

  ```
  class Point {
  double x_coord, y_coord;
  public:
  Point() : x_coord(0), y_coord(0) {} 
  Point(double x, double y) : x_coord(x), y_coord(y) {} 
  double getX() const { return x_coord; } 
  void setX(double x) { x_coord = x; } 
  ...
  }
  ```

- Want to have points with **int** coordinates, or **float** coordinates, or ...

- Let’s make it a template!
The **Point** Class-Template

```cpp
template<typename T> class Point {
    T x_coord, y_coord;
public:
    Point() : x_coord(0), y_coord(0) {}
    Point(T x, T y) : x_coord(x), y_coord(y) {}
    T getX() const { return x_coord; }
    void setX(T x) { x_coord = x; }
...
};
```

- **Parameterized on coordinate-type, named** \( T \)
  - Instead of `double`, just say \( T \) instead
  - “**typename** \( T \)” means any type – primitives or classes
  - Can use `class` instead of `typename` (means same thing)
    - “**class**” is a bit confusing since it also allows primitives
Where Templates Live

- Templates generally live *entirely* in `.hh` files
  - Unlike normal classes, *no* code in `.cc` files
  - Code that uses a template must see the entire template definition
  - C++ compilers treat them like big macros…
    - …with type-checking and many other capabilities.

- So, Point template goes into `Point.hh`
  - *All* Point code goes into `Point.hh`
  - No more `Point.cc`, now that it’s a template
Using Our **Point** Template

- Using the template is just as easy:
  
  ```
  Point<float> pF(3.2, 5.1);  // Float coordinates
  ```
  - “T means `float` everywhere in `Point` template”
  - The class’ name is `Point<float>`

- Now we want a **Point** with integers
  
  ```
  Point<int> pInt(15, 6);  // Integer coordinates
  ```
  - “T means `int` everywhere in `Point` template”
  - The class’ name is `Point<int>`

- C++ makes a whole new class for each unique template instantiation
What Types Can **Point** Use?

```cpp
template<typename T>
class Point {
    T x_coord, y_coord;

public:
    Point() : x_coord(0), y_coord(0) {}
    Point(T x, T y) : x_coord(x), y_coord(y) {}
    T getX() const { return x_coord; }
    void setX(T x) { x_coord = x; }
    ...
};
```

- In *this* template, can only use certain types for **T**!
  - **T** must support initialization to 0
  - **T** must support copy-construction
  - **T** must support assignment
Enhancing the **Point** Template

- Now let’s add the `distanceTo()` function.
  ```cpp
template<typename T>
class Point {
  ...
  T distanceTo(const Point<T> &other) const {
    T dx = x_coord - other.getX();
    T dy = y_coord - other.getY();
    return (T) sqrt((double) (dx * dx + dy * dy));
  }
};
```

- Now, *what else* must `T` support??
  - Addition, subtraction, and multiplication!
  - Casting from `T` to `double`, and casting from `double` to `T`
  - The types we can use for `T` are pretty constrained now.
Template Gotcha #1

- A major problem with templates:
  - You can’t *explicitly* state the requirements of what operations $T$ must support.

- If someone uses a template with a type that doesn’t support the required operations:
  - You just see a bunch of cryptic compiler errors.

- When you write templates:
  - Clearly document what operations the template-parameters must support.

- If you use STL much, you will learn these things. 😊
Other Template Parameters

- Can parameterize on constant values
  
  ```cpp
  template<int size> class CircularQueue {
    char buf[size]; // Static allocation
    int head, tail;
  public:
    CircularQueue() : head(0), tail(0) {}
    ...
  };
  ```
  
- `size` is a constant value, known at compile-time
- Can declare different size circular queues
  
  ```cpp
  CircularQueue<1024> bigbuf;
  CircularQueue<16>  tinybuf;
  ```
  
- No dynamic memory management
  - Faster, smaller, easier to maintain
Multiple Template Parameters

- Can specify multiple parameters
  ```cpp
template<typename T, int size>
class CircularQueue {
    T buffer[size];
    int head, tail;
    ...
};
```
- Parameters can also refer to previous parameters
  ```cpp
template<typename T, T default> class SparseVector {
    ...
    // Return value at index, or default value.
    T getElem(int index) {...}
};
```
- Now 0 doesn’t have to be the default value
Large Functions in Templates

- Sometimes template-class functions are big
  - Can put them after template-class declaration
  - (Just like an inline member function declaration)
  - Must still declare like a template

```cpp
template<typename T> class Point {
    ...  
    T distanceTo(const Point<T> &other) const;
};
```

```cpp
template<typename T> inline
T Point<T>::distanceTo(const Point<T> &other) const {
    ...
}
```
A template param can be another template

A vector of 20 Points:

```cpp
vector<Point<float> > pointVect(20);
```

Note the space between two `>` characters!

Compilers usually barf if there isn’t a space:

```cpp
vector<Point<float>> pointVect(20);    // BAD!!!
```

g++ is nice about it, though:

```cpp
error: `>>' should be `> >' within a nested template argument list
```
This Week’s Lab

- Create a C++ class template from a collection written in C
- Report illegal uses by throwing exceptions
  - Internally, code will still use assertions to check for correctness issues
- Write some simple test code
  - Do some basic testing of template’s correctness
  - Verify exception-handling behavior with test code
  - Try storing objects in your template too