Chapter 10

Exceptions

Error Handling in C++
Some Error Handling Styles

There are several different ways a function may react when encountering an unexpected/erroneous situation:

- **Return an impossible value:**
  ```c
  Set SetCreate(/* args */); // ok: returns NULL on error
  int atoi(const char *); // What to return for "bad string"?
  ```

- **Return a special type:**
  ```c
  Result sqrt(double arg, double *res); // No composition 🙁
  ```

- **The user reports an interest:**
  ```c
  double sqrt(double, Result *r); // r may be NULL
  ```

- **Provide a global mechanism:**
  ```c
  E.g, errno in C (should we have a single one? Many of them?)
  ```

- **Let an object hold the status:**
  ```c
  E.g, cin in C++
  ```

- **Roll back to a “safer” state:** The crude *setjmp and longjmp* in C
Some Error Handling Styles (cont.)

One common drawback with all
Error handling is local

The two items using objects need explanations

• The global mechanism loses information if not inquired immediately

• The status within an object can be retrieved just as long as this object is “alive”
  – It works for cin because it is a global object
The Challenge in Handling Errors

• In every decent software there are cases where the function that encounters an unexpected situation has no clue about an appropriate reaction, nor does the function that called it.

• This is the case for practically all library functions.

• The consequence is that an error-value should propagate through a series of function-calls, where each call should be wrapped with error-handling code

E.g, in our Graph:

RoadMap() => GraphCreate() => SetCreate() => malloc()
The Challenge in Handling Errors (cont.)

- There are circumstances in C++ where none of the aforementioned error reporting styles can work:
  - In constructors (no return value and Result argument is futile)
  - In overloaded operators
    » How to report an error at
      \[
      \text{Rational } r = (a + b) \times (c + d/e);
      \]

- Note that error handling includes resource release before leaving a scope
  - No memory freeing in C
  - No closing files in Java
  - C++ provides practically anything via destructors
Error Handling is a Two-Stage Operation

- The implementation (supplier code) knows how to detect an error.
- The application (user code) knows how to properly handle an error.

```cpp
int string2number(const string& s)
{
    if (s == "")    // . . .
    int result(0), i(0);
    if (is_sign(s[i])) {++i;/*..*/}
    if (s[i] == ".")  // . . .
    if (!isdigit(s[i])// . . .
    // . . .
    return result;
}
```

### Application

```
vector<double> Solve()
{
    Polynomial<int> p;
    cin >> p;
    return p.solve();
}
```

### Implementation

```
int string2number(const string& s)
{
    if (s == "")    // . . .
    int result(0), i(0);
    if (is_sign(s[i])) {++i;/*..*/}
    if (s[i] == ".")  // . . .
    if (!isdigit(s[i])// . . .
    // . . .
    return result;
}
```

- The functions between them are better be **ignorant** to such errors.
The Throw-Catch Game

- Error handling is a path parallel to the return path
- When a throw is activated, stack unwinding occurs and destructors are activated

- If, during a throw, a destructor throws additional exception `std::terminate()` is called – we cannot handle two exceptions
The Throw-Catch Game (first try)

- The implementation (supplier code) knows how to detect an error and throws appropriate exceptions.
- The application (user code) knows how to properly handle an error by catching those it can.

**Application**

```cpp
int main()
{
    try {
        vector<int> v = Solve();
        cout << v << endl;
    }
    catch (const string& s)
    {
        if (s == "empty")..
        if (s == "dot") ..
        if (s == "bad_ch").
    }
}
```

**Implementation**

```cpp
int string2number(const string& s)
{
    if (/* */) throw string("empty");
    int result(0), i(0);
    if (is_sign(s[i])) {++i; /*..*/}
    if (/* */) throw string("dot");
    if (/* */) throw string("bad_ch");
    // . . .
    return result;
}
```

- Middle functions intervention is eliminated
- Handling different errors is still cohesive
The Throw-Catch Game (second try)

Application

```cpp
using namespace string_err;
int main()
try {
    vector<int>
        v = Solve();
    cout << v << endl;
    return 0;
}
catch(const empty& e) {
    // Do something
}
catch(const dot& d) {
    // Do something else
}
catch(const bad_ch& b) {
    // Do other stuff
}
```

What did we gain?

Preparation

```cpp
namespace string_err {
    class empty {};  
    class dot {};   
    class bad_ch {};
}
```

Implementation

```cpp
int string2number(const string& s)
{ using namespace string_err;
    if (/**/ throw empty());
    int result(0), i(0);
    if (is_sign(s[i])) {++i;/**..*/}
    if (/**/ throw dot());
    if (/**/ throw bad_ch());
    // . . .
    return result;
}
```
The Throw-Catch Game (second try cont.)

Once catches are not cohesive they can be moved around

**Application**

```cpp
namespace string_err {
    class empty {};
    class dot {};
}

vector<double> Solve()
    { Polynomial<int> p;
      try {
        cin >> p;
      } catch(const empty& e) {
        return vector<double>();
      }
      return p.solve();
    }

if (/**/)
    throw dot();
if (/**/)
    throw bad_ch();
//...
```
The Throw-Catch Game (third try)

A base class catch will be activated for any derived one

```cpp
int main()
try {
    vector<int>
    v = Solve();
    cout << v << endl;
    return 0;
}
catch(const dot& d) {
    // Do something else
}
catch(const bad_ch& b) {
    // Do other stuff
}
catch(const str_err&s) {
    // A general response
}
catch(...) {
    // Ultimate generality
}
}

class str_err {};
class empty : public str_err {};
class dot : public str_err {};
class bad_ch : public str_err {};

try {
    cin >> p;
} catch(const empty& e) {
    return vector<double>();
}
return p.solve();

if (/**/**) throw dot();
if (/**/**) throw bad_ch();
// . . .
return result;
```
A good practice is to publicly derive every exception from `std::exception` class.

```cpp
int main()
try {
    vector<int>
    v = Solve();
    cout << v << endl;
    return 0;
}
catch(const dot& d) {
    // Do something else
}catch(const bad_ch& b) {
    // Do other stuff
}catch(const str_err& s) {
    // A general response
}catch(exception& e) {
    // Ultimate generality
}
```
Error Handling at Constructors

- A constructor may throw an exception during initialization

```cpp
String::String(char const *t)
try:  len(len_eval(t)), s(strcpy(new char[len+1],t) {})
catch(NullPtr& e) {
   /* . . . */ throw;
}
catch(std::bad_alloc& e) {
   cerr << e.what;   throw;
}
```

- A destructor **should not** throw an exception although it can

The only code that is activated while a `throw` is rolling back is destructors’
If a Dtor throws an exception while another one is active, the program aborts.
**Exception Specification (deprecated)**

`noexcept/noexcept(const-expr)` is the replacement

- A function may provide information about exceptions that it may throw (directly or indirectly)
  
  ```
  int f() throw(int,bad_alloc); // May throw derived
  double g() throw(); // Will throw none
  void h(); // May throw anything
  ```

- These specifications are checked at **runtime**
- If a specification is broken,
  - the function `unexpected()` is called
- By default, `unexpected()` calls `terminate()`
- During such a process `bad_exception` may be thrown
- A destructor should not throw while an exception is active
  - if it does, `std::terminate()` is called
Using Exceptions Sensibly

- **Basic Guarantee**: No resources leaked,
  - object(s) are still usable
  - consistent but not necessarily predictable state
- **Strong Guarantee**: Program state remains unchanged
- **Nothrow Guarantee**: The function will not emit an exception (e.g. swap())
- These specifications are checked at runtime
- If a specification is broken, the function `unexpected()` is called
- By default, `unexpected()` calls `terminate()`
- During such a process `bad_exception` may be thrown
- A destructor should not throw while an exception is active
  - if it does, `std::terminate()` is called
More?

Take the course

Object Oriented Programming after Programming languages