Exceptions

Error Handling in C++
Error-Handling Mechanisms

A few ways a function may react when it encounters an unexpected/erroneous situation:

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

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

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

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

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

- **Roll back to a “safer” state:** The crude setjmp/longjmp in C
Primitive C Mechanisms

- C’s `errno`: A global variable, updated by any system function. Thus, any error overrides the previous one.
- Any user compilation unit may define
  
  ```c
  extern int errno;
  ```
- Since any modification of `errno` erases its previous value, errors need to be handles (or at least identified) immediately.
- A primitive exception handler:
  - Save local state:
    ```c
    Call int setjmp (jmp_buf env); // returns 0
    ```
  - Restore (somewhere else) the same state:
    ```c
    Call void longjmp (jmp_buf env, int value);
    ```
  - Execution resumes as if `setjmp()` has just returned. If `value` is 0, then `setjmp()` “returns” 1; otherwise it returns `value`. 
Error-Handling Mechanisms (cont.)

A common **drawback**: Error handling is **local**

**Issue about using objects:**

- A global mechanism loses information if information is not inquired immediately
- The status within an object can be retrieved just as long as the object is “alive”
  - It works for `cin` because it is a global object
The Challenge

- In large software packages the function that encounters an unexpected situation (or even the function that called it) has no clue about the appropriate reaction.
- This is the case for practically all library functions.
- The consequence is that an error-value should propagate through a series of function calls, and each call should be wrapped with error-handling code.
In C++, sometimes none of the methods mentioned above can work:

- **In constructors** (there is no return value, and a `Result` argument is futile)
- **In overloaded operators**

How can an error in

```
Rational r = (a+b) * (c+d/e);
```

be reported?
The Challenge (cont.)

- Error handling also requires resource releasing before leaving a scope
  - No dynamic memory freeing in C
  - No closing files in C/Java
  - C++ provides a working solution for the majority of cases via destructors called for automatic variables

Introduction to Systems Programming
**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;
    
    return p.solve();
}
```

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

- `operator>>()`  
  ```cpp
  operator>>() for p calls string2number() indirectly.
  ```

- Application
- Implementation

**The functions between them are better be ignorant to such errors**
The error-handling path is *parallel* to the call-return path.

When a throw is activated, *stack unwinding* occurs while calling the *destructors*.

- If, during a throw, a destructor throws an additional exception, then `std::terminate()` is called, since two exceptions cannot be handled simultaneously!
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

```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").
    }
}
```

```cpp
int string2number(const string& s)
{
    vector<int> v = Solve();
    cout << v << endl;
    if (is_sign(s[i])) ++i; /*..*/
    if (s == "empty") ..
    if (s == "dot") ..
    if (s == "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
}
```

**Preparation**

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

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;
}
```

**Implementation**

**What did we gain?**
The Throw-Catch Game (second try, cont.)

Once catches are not cohesive, they can be moved around

using namespace string_err;

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
}
return result;

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

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();
// . . .
return result;
The Throw-Catch Game (third try)

A base class catch will be activated for any derived one

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

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
}
```

Implementation

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

vector<double> Solve()
{ Polynomial<int> p;
  try {
    cin >> p;
  } catch(const empty& e) {
    return vector<double>();
  } catch(const bad_ch& b) {
    return p.solve();
  } catch(const str_err&s) {
    // A general response
  } catch(....) {
    // Ultimate generality
  }
}
```

Application

```
vector<int>
v = Solve();
cout << v << endl;
return 0;
```

Preparation

```cpp
// namespaces are still useful
```
A good practice is to publicly derive every exception from std::exception.

```cpp
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
}
```

```cpp
Implementation

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

```cpp
Preparation

class str_err: public std::exception {};
class empty: public str_err {};
class dot: public str_err {};
class bad_ch: public str_err {};
```

A good practice is to publicly derive every exception from std::exception.
A constructor may throw an exception during initialization (note the syntax of `try`):

```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;
    }
```
Error Handling in Destructors

- The only code that is activated while a `throw` is rolling back is that of destructors.
- A destructor syntactically can, but **should not** throw an exception!
  - If a Destructor throws an exception while another destructor is active, **the program aborts**.
Exception Specification (old, deprecated)

- A function could provide information about exceptions that it may throw (directly or indirectly)

```c
int f() throw(int, bad_alloc);

// May throw these and derived exceptions

double g() throw();  // Will throw none

void h();            // May throw anything
```
Exception Specification (since 2011)

New syntax introduced:

- Cannot relate to specific exception types.
- Can throw any exception: `void f() noexcept(false);`
- Not throwing any exception: `int f() noexcept;` or `int f() noexcept(true);` or (still) `int f() throw();`
- If a virtual function is non-throwing, then all declarations, including the definition, of every overriding function must be non-throwing as well, unless the overrider is defined as deleted:

```cpp
struct B {
    virtual void f() noexcept;
    virtual void g();
    virtual void h() noexcept = delete;
};

struct D: B {
    void f();              // Error
    void g() noexcept;     // OK
    void h() = delete;     // OK
};
```
Using Exceptions Sensibly

- Basic Guarantee: No resources leak (if DTORs free them well)
  - Object(s) are still usable
  - Consistent but not necessarily predictable state
- Strong guarantee: Program state remains unchanged
- Specifications are checked at runtime
- If a specification is broken, `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