# COMP2004
# Programming Practice
# 2002 Summer School

Kevin Pulo
School of Information Technologies
University of Sydney

## Exception Safety

- Exceptions make programming harder
- Your code should be exception safe
- It's not a matter of just using try/catch
- It's part design
- It's part minimising assumptions

## When to throw

- Your functions should offer one of the following:
- Basic Guarantee
  - Resources are not leaked
  - Objects are still usable if not predictable
- Strong Guarantee
  - Program state is as before the call
- Nothrow
  - The function will never throw

## When to catch

- Can the code handle the error and clean up?
- Is this the best place to handle it?
- Use RAII whenever it is possible
  - We'll get to this in a minute...

## Unexpected Exceptions

- What happens if a function you call throws?
- You must make sure nothing leaks
- You must maintain invariants
- This is easier if all functions offer one the guarantees

## Simple Throw Guides

- Throw when you can not handle the error
- Document those errors
- Document the guarantee given

# Simple try/catch Guides

- Use to handle errors you can deal with
  - Use RAII as much as possible
- Use to translate an exception
  - From low level to high level for example
- catch(...) to prevent exception leakage
  - Only when caller code can't handle exceptions

# RAII

- Resource Acquisition is Initialisation
- A C++ idiom for dealing with resources
- Uses automatic variables to handle resources
  - Since the language manages them for you

# Bad Example Code

```
void some_function(int size) {
    char *fred = new char[size];
    // do some stuff
    delete [] fred;
}
```

- What happens if an exception is thrown?
- The memory resource is leaked!

# Fixing with try/catch

```
void some_function(int size) {
    char *fred = new char[size];
    try {
        // do some stuff
    } catch (...) {
        delete [] fred;
        throw;
    }
    delete [] fred;
}
```

# In General

```
void some_function() {
    // acquire resource A
    // do stuff
    // acquire resource B
    // do stuff
    // possibly more resources...
    // release resource B
    // release resource A
}
```

# Too Complicated

- Using try/catch blocks is too hard
  - Lots of duplicate code
  - Lots of exception handling run-time overhead
  - Verbose and tedious, error prone
  - Doesn't scale
  - Results in brittle code

# Fixing with local variable

- Would like to do

```
void some_function(int size) {
        char fred[size];
        // do some stuff
}
```

- If an exception is thrown, fred is automatically deleted when the function ends
- But can't do this, since size isn't known until runtime

# Use a class/struct

```
struct char_array {
        char *array;
        char_array(int size) {
                array = new char[size];
        }
        ~char_array() {
                delete [ ] array;
        }
        operator char*() {
                return array;
        }
};
```

# Original now becomes

```
void some_function(int size) {
        char_array fred(size);
        // do some stuff
}
```

- If an exception is thrown fred will be destroyed
  - Because it is an automatic variable
- Its destructor will be called
- Thus the array will be deleted

# auto_ptr

- Templated library class
- #include <memory>
- Is a wrapper around a pointer
- Can be dereferenced like the pointer
- Destructor deletes the object pointed to

# auto_ptr example

```
void do_enrolment(string name,
                                 string course) {
        Person *pp = new Person(name);
        auto_ptr<Person> p(pp);

        p->enrol(course);
        // do some stuff
        if (error_found) throw Error();
        // do some more stuff
        // not needed: delete pp;
}
```

# auto_ptr caveats

- auto_ptr's not always that easy
- Copying an auto_ptr leaves the original pointing nowhere
  - Thus can't copy a const auto_ptr
- Shouldn't have > 1 auto_ptr to an object
  - Object may be deleted twice
- Shouldn't use in containers
  - vector< auto_ptr<Person> > &v;
  - Due to copy semantics above

## auto_ptr usage

- So stick to simple usage of auto_ptr
- Exception safe automatic pointers
- As a prewritten version of the char_array struct

## Constructors and Exceptions

- Constructors can throw exceptions
- Keep this in mind when writing C++
- If so, no object is constructed
- Is the usual way to indicate an error
  - Since constructors can't return anything

## Member Initialiser Exceptions

```
SomeClass::SomeClass(int size)
: vec(size)
{
    // Contents of constructor...
}
```

- What if vec(size) throws an exception?
  - Passed on to caller
  - Constructor can catch
  - Syntax is a little bizarre

## Member Initialiser Exceptions

```
SomeClass::SomeClass(int size)
try
    : vec(size)
{
    // Contents of constructor...
}
catch (Error e) {
    // ...
}
```

## Copy Constructors and Exceptions

- Copy constructors are a bit different
  - They can throw exceptions
  - But generally shouldn't
  - The library assumes they don't
- Same for the assignment operator

## Destructors and Exceptions

- Throwing an exception in a destructor is risky
- Automatic variable destructors
  - Are called during stack unwinding
  - Which is part of exception handling
  - Throwing an exception then will terminate() the program
- Best to stick to exceptions in constructors only (where possible)