

Programming in C++

Session 10 – When things go wrong: Exceptions and Resource management

Dr Christos Kloukinas

City, UoL

<https://staff.city.ac.uk/c.kloukinas/cpp>
(slides originally produced by Dr Ross Paterson)



Outline

- Exceptions in C++
- **Resource acquisition is initialization (RAII)**
 - A **fundamental** C++ technique
 - Ensures that resources are freed, even in the presence of exceptions, without writing lots of exception-handling code.
- RAII: a special case of the *smart pointer* and *proxy* patterns

- Plus Revision!

Part I

Exceptions

Failures (revision)

- Method cannot meet its specification?
⇒ Communicate this to its caller!
- May cause the caller to fail, and so on
But sometimes the caller can work around the failure
- Might be necessary to clean up in the event of failure
- Traditional (C) approach – an `if` on a status variable – is very cumbersome (and often left out)
- Disciplined use of *exceptions* makes error-handling clearer and more robust

Throwing an exception in C++

- Objects of any class can be thrown (even basic types):
`class my_exception { ... };`
- The `throw` statement typically takes a TEMPORARY OBJECT:
`throw my_exception("Bad date");`
- The exception should be caught by reference.
 - This is the "best practice"
 - Can also be caught by value.
But avoid it, since catch-by-value:
 - Slices derived exceptions
 - Requires copying

Catching an exception in C++

- C++ has a `try/catch` statement, largely copied by Java:

```
try {  
    // do something that might fail  
} catch (my_exception &e) {  
    // deal with the exception  
} catch (AnotherException) {  
    // deal with the exception  
}
```
- Like Java, exceptions may form hierarchies
 - A `catch` clause also handles any derived classes
- C++ has no `finally` clause

The C++ treatment of exceptions

- If (inside a `try` block
&& there's a matching `catch` clause)
Then execute the first matching `catch` clause
- "matching" = the exception type or some base type of it
- Otherwise
 - Exit from the current block or function
Destroying any locally allocated variables in the process, and
 - Continue searching for a matching `try` block
- If the `main` function is exited in this way
Halt the program with an error message.

This is called ***unwinding the stack***

Clean up and rethrow

Often exception handlers are used to free resources on failure:

```
// acquire resource  
try {  
    // do something that might fail  
    // free resource  
} catch (...) { // any exception  
    // free resource  
    throw; // rethrow the exception  
}
```

This can often be avoided, using the RAII technique:
"Resource Acquisition Is Initialization".

Note on syntax:

- Catch any exception: `catch (...)`
- Rethrow an exception: `throw;`

Resource management

- Programs acquire resources
Allocate memory, open files, create windows, acquire locks, etc.
- These resources should be released – Even if there are exceptions!
- Most resources are freed when a program terminates :-)
- But some are not, *e.g.*, some kinds of lock :- (
- Releasing resources properly is tricky and easy to get wrong

A typical pattern of resource use

Resources must often be released in the opposite order to acquisition:

```
// acquire resource 1
// ...
// acquire resource n

// use resources

// release resource n
// ...
// release resource 1
```

Just like locally allocated data!

Resource acquisition is initialization (RAII)

Introduce a resource management class with

- A constructor to acquire the resource (or just to record it)
- A destructor to release the resource
- Possibly an access method

Locally allocate an object of this class when acquiring the resource, and the resource will be **automatically** released!

Moreover, resources will be released in the correct order!

```
// Without RAII :- (           // With RAII :-) :-)
// acquire resource           {
try {                          // acquire resource
    // this might fail        try {
    // now free resource       // this might fail
} catch (...) { //any exception }
    // free resource          } // resource freed here!
    throw; //rethrow exception
}                               // A try in main is enough!
```

Example: file streams

- `ifstream/ofstream`'s constructors open streams
`ifstream in("file.txt");`
- Their destructors close the streams
(though one can do it earlier if required)
- Hence code safely like this:

```
{
    ifstream inp("file.txt");
    // read and process file
} // inp is destroyed here (IF inside a try{}!!!)
```

Whether control leaves the block normally or due to an exception, the file stream will be closed.
(*must be a surrounding try somewhere!*)

Example: storage management

This class manages the deletion of dynamically allocated `point` objects

```
class point_manager {
    point *ptr;

public:
    point_manager(point *p) : ptr(p) {}

    ~point_manager() { delete ptr; }
    point_manager(const point_manager &) = delete;
    point_manager &operator=(const point_manager &)
        = delete;
};
```

Using the `point_manager`

Whenever a `point` that is only required for this block is dynamically allocated, make a local `point_manager` to manage it:

```
point *p1 = new point(20,30);
point_manager m1(p1);

point *p2 = window->get_middle();
point_manager m2(p2);
```

On leaving the block (normally, via `return`, or by an exception), then `m2` will be destroyed, which will `delete p2`, and then `m1`, which will `delete p1`.

Generic storage management

The standard header `<memory>` **provided [*]** a class `auto_ptr`. Here is a simplified version:

```
template <typename T> class auto_ptr {
    T *_ptr;

public:
    auto_ptr(T *ptr) : _ptr(ptr) {}

    ~auto_ptr() { delete _ptr; }
};
```

(more to come later)

[*] Until C++11 – deprecated since!!!

Using `auto_ptr` – The promise

- To ensure that dynamically allocated storage is reclaimed, create a local `auto_ptr` to manage it:

```
point *p = new point(20,30);
auto_ptr<point> p_ptr(p);
```

- On leaving the block, `p` is automatically deleted.
- One can also use `auto_ptr` as a subobject
No need to write our own destructors!
- Since all methods are inline, there is very little overhead.

More convenience

We add the following operator definitions to the `auto_ptr` class:

```
T & operator*() { return *_ptr; }
T * operator->() { return _ptr; }
```

Then we can use the `auto_ptr` as a *proxy* for the pointer:

```
auto_ptr<int> ip(new int);
*ip = 3;

auto_ptr<point> pp(new point(20,30));
pp->x = 4;
pp->y = 5;
```

Completing `auto_ptr`

Gang of Three!

- Since `auto_ptr` has a non-trivial destructor, it requires
 - A copy constructor; and
 - An assignment operator
- **Only one of the copies of an `auto_ptr` should call `delete`.**
- Might as well add a default constructor too.

Let's do it!

```
template <typename T>
auto_ptr( ) : _ptr(nullptr) {}

template <typename T>
auto_ptr( auto_ptr<T> & o ) { // XXX No const & !!!
    _ptr = o._ptr;
    o._ptr = nullptr;      // XXX other loses pointer!
}
```

Completing `auto_ptr` – II

```
template <typename T>
auto_ptr<T> &
operator=( auto_ptr<T> & o ) { // XXX No const & !!!
    if (this != &o) {
        delete _ptr;
        _ptr = o._ptr;
        o._ptr = nullptr;      // XXX o loses pointer!
    }
    return *this;
}
```

(Smart pointers

`auto_ptr` is a so-called “*smart pointer*”

It looks like a pointer, but does something extra

Some other examples:

reference counting proxy counts references to a dynamically allocated object, and deletes it when count reaches zero

persistent data proxy reads data from a file on first use, and saves it in the file on destruction

virtual/lazy object proxy delays creating a complex object until it is used (and if the object is never used, avoids creating it)

The Proxy pattern)

More generally, a *proxy* is any object that is interposed between the client and some other object. Some other uses:

- wrapper** proxy provides consistent access to foreign language data
- protection** proxy provides more limited access to the object, for greater security
- handle** proxy represents an object in a different address space, e.g., an operating system object, a graphical system object, or an object on another machine

...

*May you live
in interesting times... :- (*

(2019: This 2011 statement did not age well at all!)

C++11

- auto_ptr** deletes its pointer using `delete` !
 - So cannot manage a pointer to an array (needs `delete[]`)
- auto_ptr**'s "copy" constructor **steals** the other object's pointer!
 - That's not copying, that's moving! (*polite version of "stealing"*)
 - So cannot use `auto_ptr` inside STL containers (*containers think they copy elements when they don't*)
- C++11: Use `unique_ptr` instead (or `shared_ptr`)
 - `unique_ptr` offers a **move** constructor but no copy constructor:

```
unique_ptr(unique_ptr<T> && x); // rvalue reference...
unique_ptr(unique_ptr<T> & x) = delete; // reference...
```

- You need to know how `auto_ptr` works, as old code uses it (BUG!)
- And to understand "rvalue references" (*and why we need them*)
- You need to learn the others for your coding
- These also work with arrays by the way:

```
unique_ptr<int []> array(new int [30]);
```

Programming in C++

2023-12-12

C++11

```
C++11
auto_ptr deleted its pointer using delete !
- So cannot manage a pointer to an array (needs delete[])
auto_ptr's "copy" constructor steals the other object's pointer
- That's not copying, that's moving (polite version of "stealing")
- So cannot use auto_ptr inside STL containers (containers think they copy elements when they don't)
C++11: Use unique_ptr instead (or shared_ptr)
- unique_ptr offers a move constructor but no copy constructor
unique_ptr(unique_ptr<T> && x); // rvalue reference...
unique_ptr(unique_ptr<T> & x) = delete; // reference...
You need to know how auto_ptr works, as old code uses it (BUG!)
And to understand "rvalue references" (and why we need them)
You need to learn the others for your coding
These also work with arrays by the way:
unique_ptr<int []> array(new int [30]);
```

C++11 – II

Advanced – not in the exam (neither is `unique_ptr` nor rvalue references/move constructors).

- shared_ptr**:
 - "It's complicated" (see [stackoverflow bit.ly/1SiGPyc](https://stackoverflow.com/questions/11111111/1SiGPyc))
 - And the class documentation:
https://en.cppreference.com/w/cpp/memory/shared_ptr
 - Especially the constructors:
https://en.cppreference.com/w/cpp/memory/shared_ptr/shared_ptr
- !!! Avoid temporary smart pointers.
Why? See Boost bit.ly/1PYanO3
- Or BETTER YET use `make_shared`
(tricky... – stackoverflow bit.ly/1KdK2ao)

Further reading

- Exceptions: Stroustrup 14, Meyer 12.
- Resource acquisition is initialization (RAII): Stroustrup 14.4.
- Smart pointers: Stroustrup 14.4.2, 11.10.

- Check out on StackOverflow the iterator proxy I created for implementing `copy_if_and_transform`
<https://stackoverflow.com/questions/23579832/why-is-there-no-transform-if-in-the-c-standard-library/74288551#74288551>
or <https://bit.ly/3Yd0dSM>
(it tries to make `*from` behave differently, depending on the context)

Part II

Revision

Major Differences between Java and C++

C++ allows direct access to objects!!!

- **[*]** call-by-value & call-by-reference (session 1 and since)
- operator overloading (session 3)
- genericity or template classes (sessions 4–6)
- **[*]** slicing of derived objects on copying (session 6)
- memory management
 - local allocation of objects (sessions 1–2 and since, esp. 9–10)
 - pointers (sessions 5 and 6)
 - dynamic allocation (sessions 8–9)
- multiple inheritance (session 7)
- **[*]** gang of three (session 8)
- **[*]** Rvalue references (& call-by-rvalue-reference – session 10)

[*] Because C++ allows direct access to objects...

Things you should be able to do

- Write simple classes and functions in C++
- Use the containers and iterators of the Standard Template Library to write more compact (& correct!) programs
- Understand the difference between call-by-value and call-by-reference
- Appreciate the various meanings of `const` in C++, and know when to use them
- Read programs using overloaded operators, by identifying which methods or independent functions are called
- Define overloaded operators for new types
 - As member functions
 - As independent functions

(continued)

More things you should be able to do

- Distinguish between objects and pointers (& how each behaves)
- Know how to use static, local, dynamic and temporary allocation, appreciating their properties and distinctive features
- Understand the properties of subobjects (= fields of other objects)
- Use inheritance, method redefinition and abstract classes in C++
 - Know the order of initialisation (parents [*], fields [*], constructor) and destruction (opposite) [*] IN THE ORDER OF DECLARATION!!!!

BE CAREFUL WITH FIELD INITIALISATION!!!

- Write generic classes and functions in C++
- And use the standard generic algorithms!!!

(continued)

Even more things you should be able to do

- Use multiple inheritance in C++, knowing how to specify replicated vs. virtual inheritance (**virtual**)
- Explain — Gang of Three
 - 1 What the automatically generated constructors, destructors and assignment operators do
 - 2 When they are inadequate, and if so
 - 3 How they should be replaced
- Use the exception syntax of C++ (**try, catch, throw, rethrow**)
- Use RAII (“resource acquisition is initialization”) to safely release resources, even in the presence of exceptions
 - Use **unique_ptr** (and less often **shared_ptr** [*]) to automatically manage your pointers
[*] sharing makes it harder to parallelise)

└ Even more things you should be able to do

Empty On Purpose

Even more things you should be able to do

- Use multiple inheritance in C++, knowing how to specify replicated vs. virtual inheritance (**virtual**)
- Explain — Gang of Three
 - 1 What the automatically generated constructors, destructors and assignment operators do
 - 2 When they are inadequate, and if so
 - 3 How they should be replaced
- Use the exception syntax of C++ (**try, catch, throw, rethrow**)
- Use RAII (“resource acquisition is initialization”) to safely release resources, even in the presence of exceptions
 - Use **unique_ptr** (and less often **shared_ptr** [*]) to automatically manage your pointers
[*] sharing makes it harder to parallelise)

└ Even more things you should be able to do

Empty On Purpose

Even more things you should be able to do

- Use multiple inheritance in C++, knowing how to specify replicated vs. virtual inheritance (**virtual**)
- Explain — Gang of Three
 - 1 What the automatically generated constructors, destructors and assignment operators do
 - 2 When they are inadequate, and if so
 - 3 How they should be replaced
- Use the exception syntax of C++ (**try, catch, throw, rethrow**)
- Use RAII (“resource acquisition is initialization”) to safely release resources, even in the presence of exceptions
 - Use **unique_ptr** (and less often **shared_ptr** [*]) to automatically manage your pointers
[*] sharing makes it harder to parallelise)

Even more things you should be able to do

Even more things you should be able to do

- Use multiple inheritance in C++, knowing how to specify replicated vs. virtual inheritance (see 4.4.4)
- Explicit — Const of This
- Using an auto-generated constructor, destructor and assignment operator
- When they are redundant, and if so
- How they should be replaced
- Use the exception syntax of C++ (try, catch, throw, noexcept)
- Use try/catch to propagate or catch/throw; to safely release resources, even in the presence of exceptions
- Use `std::move`, `std::move_if_noexcept`, `std::move_if_noexcept` to automatically manage your pointers
- (*) `std::move` makes it harder to parallelize

Final Notes – V

- What to do when you receive an exception?
You're at a family party and cousin Jim starts to choke on a piece of meat!
 - 1 Catch the exception and ignore it – hide Jim in a closet and pretend nothing's happened.
 - 2 Catch the exception and log it – “Dear diary, Jim once more ruined the party...” (after having hidden Jim in a closet).
 - 3 Catch the exception and fix the problem – Help Jim spit the piece of meat that is choking him.
 - 4 Not catch the exception but let it propagate instead to your caller (or catch/rethrow), who might know how to fix it – Call 999 and let them know there's someone choking; they'll deal with it (if they can).

HINT: It's neither #1 nor #2 that you should be doing. . .

Even more things you should be able to do

Even more things you should be able to do

- Use multiple inheritance in C++, knowing how to specify replicated vs. virtual inheritance (see 4.4.4)
- Explicit — Const of This
- Using an auto-generated constructor, destructor and assignment operator
- When they are redundant, and if so
- How they should be replaced
- Use the exception syntax of C++ (try, catch, throw, noexcept)
- Use try/catch to propagate or catch/throw; to safely release resources, even in the presence of exceptions
- Use `std::move`, `std::move_if_noexcept`, `std::move_if_noexcept` to automatically manage your pointers
- (*) `std::move` makes it harder to parallelize

Final Notes – VI

Further pointers:

- “What should I throw?”
A temporary object.
<https://isocpp.org/wiki/faq/exceptions#what-to-throw>
- “What should I catch?”
Catch by reference if given the choice (avoids copying).
<https://isocpp.org/wiki/faq/exceptions#what-to-catch>
- “But MFC seems to encourage the use of catch-by-pointer; should I do the same?” (aka When in Rome. . .)
When working with MFC yes, otherwise no as it's not clear who's responsible for deleting the pointed-to object.
<https://isocpp.org/wiki/faq/exceptions#catch-by-ptr-in-mfc>
- “What does throw; (without an exception object after the throw keyword) mean? Where would I use it?”
Re-throw.
<https://isocpp.org/wiki/faq/exceptions#throw-without-an-object>
- “How do I throw polymorphically?”
To catch derived exceptions instead of base exceptions, make sure you're throwing derived exception objects! Use virtual functions.
<https://isocpp.org/wiki/faq/exceptions#throwing-polymorphically>
- “When I throw this object, how many times will it be copied?”
Nobody knows (zero to some) but the exception object must have a copy-constructor (even if the compiler will never copy it).
<https://isocpp.org/wiki/faq/exceptions#num-copies-of-exception>