Smart Pointers, RAII

PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

Motivation

memory and resource management is hard

- memory safety
  - `new`/`delete` / `new[]`/`delete[]` pairing
  - avoid memory leaks
- deallocate exactly once
  - even with exceptions!
- common pattern for any resource

RAII

Resource Acquisition Is Initialization

- there is one class managing a resource – *owner object*
- allocates (acquires) it in constructor
- deallocates (releases) it in destructor
- sometimes the resource can be explicitly assigned/released
- usually moveable but not copyable

RAII

Resource Acquisition Is Initialization

- there is one class managing a resource – *owner object*
- allocates (acquires) it in constructor
- deallocates (releases) it in destructor
- sometimes the resource can be explicitly assigned/released
- usually moveable but not copyable

- resource can be anything
  - memory, file, socket, lock, database connection, ...
  - but the owner object owns only one

```
struct ResourceOwner {
    ResourceOwner() { /* acquire resource */ }
    ~ResourceOwner() { /* release resource */ }
};
```

RAII

Why?

- composition
  - more resources → more owner objects
  - each owner guards one resource
- ease of use
  - resource release is automatic at the end of owner object's scope
  - usually only owner objects need user-defined destructors
- heavily supported by C++ library
- exception safety
  - resource is deallocated even when exception occurs and it causes owner to go out of scope

RAII is Concise

Exception safe C++

```
void foo() {
    std::fstream f1("file1.txt");
    // work with the file ...
} // the file is closed automatically
```

Exception safe Java (6)

```
public void foo() {
    FileReader f1 = null;
    try { f1 = new FileReader("file1.txt");
        // work with the file ...
    } finally {
        try { if (f1 != null) f1.close(); }
        catch (IOException io) { }
    }
}
```

RAII is Concise

Exception safe C++

```
void foo() {
    std::fstream f1("file1.txt");
    // work with the file ...
} // the file is closed automatically
```

Exception safe Java (7+)

```
public void foo() {
    try ( FileReader f1 = new FileReader("file1.txt") )
    {
        // work with the file ...
    }
}
```

RAII – When Does It Work?

for a local owner object, the resource is
surely freed when the scope is exited:

- by `return`
- at the end of the scope
- when exception is thrown and it is caught in some scope above

surely not freed when:

- `std::exit`, `std::quick_exit`, `std::abort`, `...` is called
- `std::longjmp` is called (this is undefined behaviour!)
- a signal causes process termination
- power is turned off

may, or may not be freed (usually not):

- an exception is thrown and it is never caught (!)

RAII in STL

- `std::(i/o)fstream`
- smart pointers (`std::unique_ptr`, `std::shared_ptr`)
- containers (`std::vector` also owns memory)
- `std::lock_guard`

Smart Pointers

std::unique_ptr

- unique owner of memory
- std::unique_ptr< SomeClass >,
  std::unique_ptr< int[] > – new[] allocated
- std::make_unique< A >(a, ctor, params) (C++14)

```
void foo() {
    std::unique_ptr< int > iptr{ new int( 42 ) };
    auto x = std::make_unique< A >( 1, 8 );
}
```

Smart Pointers

`std::shared_ptr`

- shared owner, counts references (`shared_ptr` instances) for the object
- deallocates when last instance is destructed
- structure of `shared_ptr` must not contain cycles (`std::weak_ptr` to break cycles)
- copyable, copy increases reference count
- `std::make_shared< A >( a, ctor, params )`

Smart Pointers

`std::weak_ptr`

- prevent cycles from `std::shared_ptr`
- a pointer which does not own, but can detect that object is no longer alive

```
std::weak_ptr< A > wp;
{
    std::shared_ptr< A > sp = new A();
    wp = sp;

    // auto == std::shared_ptr< A >
    if ( auto locked = wp.lock() )
        locked->foo();
}
if ( wp.expired() )
    std::cout << "wp has expired" << std::endl;
```

Smart Pointers

std::enable_shared_from_this

- if you want to be able to get shared_ptr from an object (not pointer to it) it must derive from std::enable_shared_from_this
- then you can get shared pointer by shared_from_this method
- std::shared_ptr< T >(this) does not work
  - more shared pointers which do not share ownership
  - object can be deallocated more than once

```
struct X : std::enable_shared_from_this< X > {
    std::shared_ptr< X > getptr() {
        return shared_from_this();
    }
};
```

Smart Pointers: Casting

`std::static_pointer_cast`, `std::dynamic_pointer_cast`, `std::const_pointer_cast`

- to cast shared pointers, special functions are required
- the resulting `shared_ptr` shares ownership with the original

Memory in Modern C++

- never use `new`/`delete` unless writing a (low-level) library
- do not use dynamic memory if you don't have to
- use `std::unique_ptr` for objects owned by one object
  - or if ownership changes, but is never shared
  - ownership transfer by `std::move`
- use `std::shared_ptr` for shared objects
- use raw pointers to point into an object owned by someone else

Tasks

Task I – File Descriptors

Use files `05_fildes.h`, `05_fildes.cpp`

- linux/unix only
- find what are the problems of this implementation?
  - focus at resource management, exception safety
- change it to use RAII correctly and every time it is meaningful

Task II – Binary Trees

Use file `05_tree.h`

- uncover memory bugs (leaks, double/invalid free)
  - write tests, use valgrind
- fix it