Object-Oriented Programming

Iuliana Bocicor
iuliana@cs.ubbcluj.ro

Babes-Bolyai University

2018

Overview

RAII I

- **RAII** = Resource Acquisition Is Initialization.
- **Resources**
  - E.g.: memory, files, sockets, database connections.
  - Resources are *acquired* before use and then *released* after one has finished working with them (preferably, they should be released as soon as possible).
  - Failing to release a resource can cause leaks and even crashes.
  - RAII is used to *avoid resource leaks* and to write *exception-safe code*.

Example of resource leak I

```
void resourceLeak()
{
    try
    {
        int* a = new int{ 2 };
        throw std::exception{ "Hello! An exception
            has occured!\n" };
        delete a;
    }
    catch (std::exception& e)
    {
        cout << e.what();
    }
}
```

Example of resource leak II

- How can this be solved?
- One solution (workaround): clean up in the catch block. **?** Why is this not a good solution?
- Another solution: using RAII.

The idea I

- The compiler automatically calls:
  - constructors to initialize objects;
  - destructors, when the objects' scope is finished.
- When creating an object, we take responsibility for the resources in it. The constructor is responsible with resource allocation.
- The destructor does the clean up: the resource should be deallocated in the destructor.

The idea II

- As the compiler automatically calls constructors and destructors, the resource will correctly be managed.
- In this way, there will be no resource leaks.
- Advantages over garbage collection (from other programming languages):
    - RAII offers automatic management for different kinds of resources, not just memory.
    - The runtime environment is faster, as there is no separate mechanism involved (like the garbage collector).

How is it done?

- Create a wrapper for your object using resource allocation: allocation in constructor, deallocation in destructor.
- Use the wrapper object (directly) wherever you need the object.
- The resource will be deallocated when the wrapper's scope is left.

DEMO

RAII for pointers (*Lecture8_demo* - SmartPointer, SmartPointerTemplate).

RAII in STL

- You have been already using RAII!
- When an object of type ifstream or ofstream, the constructor will automatically open the file.
- When the object gets destroyed, the destructor automatically closes the file.
- The STL containers manage memory using the RAII programming idiom. Remember your dynamic vector?
- There are "smart pointers" defined in STL, which use RAII for "smart" memory management.

Smart pointers in STL I

- In modern C++, raw pointers are used only in certain cases: "small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership" (https://msdn.microsoft.com/en-us/library/hh279674.aspx).
- Smart pointers are used instead.
- Smart pointers are class templates.
- A smart pointer object is declared on the stack and initialized with a raw pointer. When it goes out of scope, its destructor is invoked.

Smart pointers in STL II

- The smart pointer owns the raw pointer ⇒ it is responsible for it (memory deallocation).
- Objects are automatically cleaned up when the smart pointers go out of scope or are set to point at something else or nothing - they get deleted when nobody is interested in them any more.
- STL smart pointers defined in the std namespace, in the header <memory>.

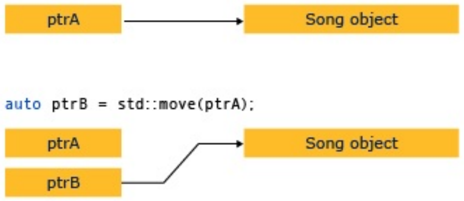
Smart pointers in STL III

- There ar 3 types of smart pointers in STL:
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr

std::unique_ptr I

- Such a smart pointer *owns its object uniquely*.
- It retains *exclusive ownership* of the object, it does not share the object.
- It is impossible for two unique_ptr objects to own the same object.

std::unique_ptr II



```
auto ptrB = std::move(ptrA);
```

Figure source: https://msdn.microsoft.com/en-us/library/hh279676.aspxl

- It cannot be copied. **?** Could such an object be passed by value?
- It can be moved to a new owner: the resource is transferred to the new owner.

std::unique_ptr III

- When it goes out of scope, the owned object is destroyed.
- It should be constructed with the make_unique function.

DEMO

unique_ptr (*Lecture8_demo_Smart_pointers_STL* - exampleUniquePtr).

std::shared_ptr I

- Retains *shared ownership* of the object.
- Several shared_ptr object may own the same object.
- Uses *reference counting*: when multiple shared pointers own the same object, these are keeping track of how many "copies" there are.
- The owned object is deleted only when the last remaining owning shared_ptr is destroyed or have given up ownership (has been reset).

std::shared_ptr II

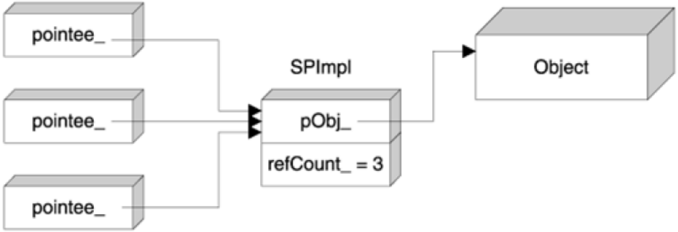


Figure source: https://stackoverflow.com/questions/9200664/how-is-the-stdtr1shared-ptr-implemented/9201435

std::shared_ptr III

- It can be copied and moved (move transfers ownership).
- shared_ptr has more overhead than unique_ptr (because of the internal reference counting), therefore, **whenever possible, prefer unique_ptr**.
- It should be constructed with the make_shared function.

DEMO

shared_ptr (*Lecture8_demo_Smart_pointers_STL* - example-SharedPtr).

std::weak_ptr I

- Used to access the underlying object of a shared_ptr without causing the reference count to be incremented.
- Is usually used to avoid dependency cycles.

std::weak_ptr II

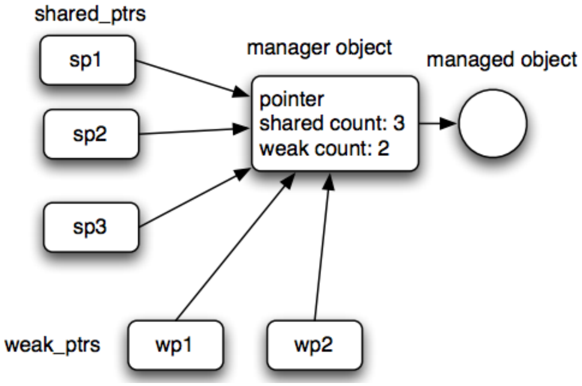


Figure source: https://ix.cs.uoregon.edu/~norris/cis330/index.cgi?n=Main.W10D1ex

std::weak_ptr III

E.g.: 2 classes - Team and Member

- A team has pointers to its members.
- Each member can have a pointer to the team it belongs to.
- **?** If all pointers (to members and to team) are shared_ptr, what happens when the team goes out of scope? (Answer: memory leak - but how and why?)
- Therefore, the members should have a weak pointer to their team.

std::weak_ptr IV

- The underying object in a weak_ptr can still be deleted even though there is a weak_ptr reference to it.
- weak_ptr can be used to create a shared_ptr.

DEMO

weak_ptr (*Lecture8_demo_Smart_pointers_STL* - teamMembersSharedPtr, exampleWeakPtr).

Advantages of smart pointers

- Smart pointes increase productivity and improve the robustness of the program.
- The programmer does not need to be concerned with memory management (provided the smart pointers are used correctly).
- They help in avoiding memory leaks and writing exception-safe code.

Smart developers use smart pointers. (Kate Gregory)

Homework I

- Write an application to keep the aircraft evidence in a country.
- Each aircraft has a **unique identifier** and a **model**, *is suitable* only for certain activities (e.g. public transportation, medical emergencies, leisure time, military) and can reach a certain *maximum altitude*.
- An aircraft can be one of the following three: helicopter, plane or hot air balloon.

Homework II

- A helicopter:
  - has the following additional characteristic: **isPrivate**, specifying whether the helicopter belongs to the state or to a private entity.
  - is suitable for activities like: military, medical emergencies, public transportation and leisure time (only if it is private).
  - can reach a maximum altitude of 12 km.
- A plane:
  - has the following additional characteristics: **isPrivate**, specifying whether the plane belongs to the state or to a private entity and **main wings** (the plane can be either monoplane or biplane).
  - is suitable for activities like: military, public transportation and leisure time (only if it is biplane).
  - can reach a maximum altitude of 26 km.

Homework III

- A hot air balloon:
  - has the following additional characteristics: **weight limit**, specifying the maximum weight limit for the balloon.
  - is suitable for activities like: leisure time.
  - can reach a maximum altitude of 21 km.
- The application should allow the following:
  - Add any type of aircraft.
  - Display all aircraft which can be used for a certain activity and save them to a file having the activity's name.
  - Diplay all aircraft which can reach at least a given altitude.