C++ - Pre-lecture 7

Smart pointers

Charanjit Kaur & Caterina Doglioni, PHYS30762 - OOP in C++ 2024

MANCHESTER
1824
The University of Manchester

Auto pointers

- Some of you have already used a feature of C++ that avoids the hassle of memory clean up in pointers
- A simple example from Wikipedia:
-
```
#include <iostream>
#include <memory>
using namespace std;

int main(int argc, char **argv)
{
    int *i = new int;
    auto_ptr<int> x(i);
    auto_ptr<int> y;

    y = x;

    cout << x.get() << endl; // Print NULL
    cout << y.get() << endl; // Print non-NULL address i

    return 0;
}
```

This is removed in C++17!!!

(So don't learn how to use it, move on to smart pointers)

- You may have been wondering why we don't tell you about them earlier…
  - this is because *raw pointers* (those seen so far) have helped you understand and practice memory management, passing by reference and by value

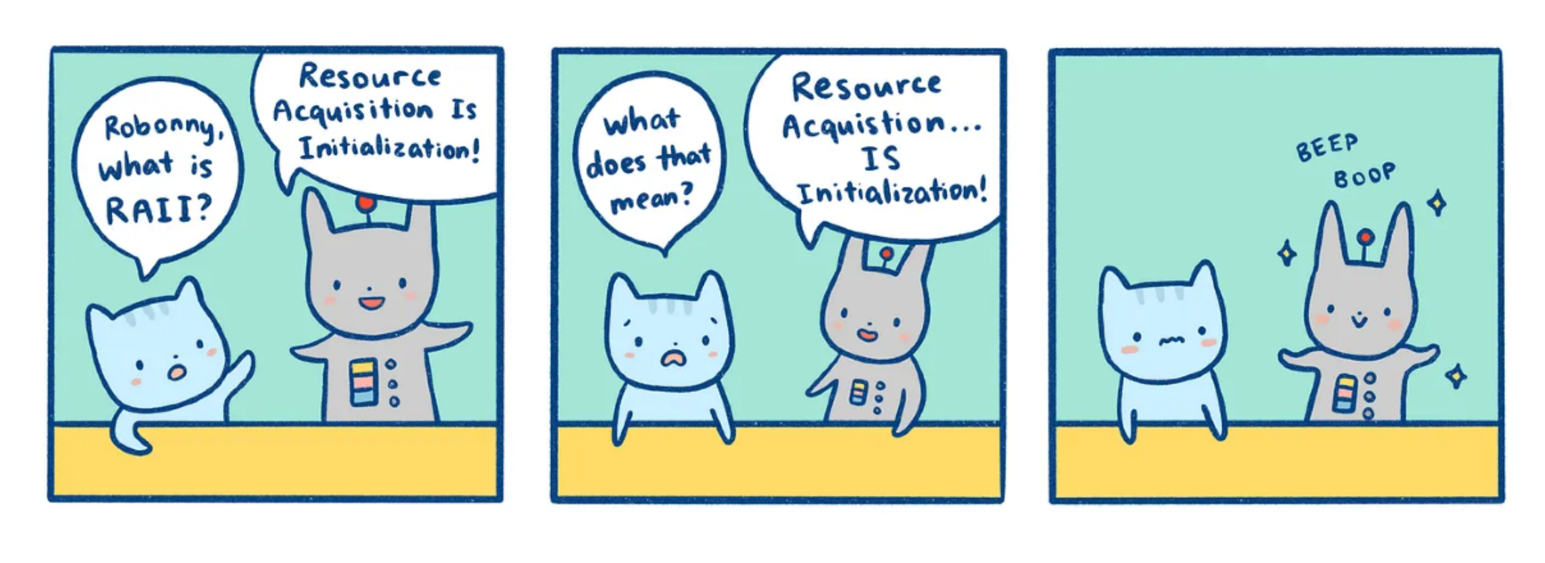
Intermezzo: C++ standards

- C++ is a language in active development: next “revision” planned for this year
- There is a **committee** that improves the language standards
- Website: https://www.open-std.org/jtc1/sc22/wg21/

- Once a standard is released, it is implemented in compilers
  - Some features also get deprecated / removed
  - E.g. auto -> smart pointers removal happened between C++11 and C++17

- How to deal with C++ standards when we write code?
  - If you’re keen, you can go through the history of C++
  - In general, a recent-enough compiler will tell you whether something you’re doing is outside new standards via errors or warnings
  - For example: g++-11 includes support for C++17
    - If you want to make sure you’re compliant with all the C++17 standards, use the flag “`c+=17`” as an element in the `args[]` vector in your *tasks.json* in VS Code
    - (this is not needed as g++-11 does that by default)
- For fun: you can compare compilers at godbolt.org [GitHub and more info]

Concept: the RAII idiom


Robonny, what is RAII?
Resource Acquisition Is Initialization!
what does that mean?
Resource Acquistion... IS Initialization!
BEEP
BOOP

https://medium.com/swlh/what-is-raii-e016d00269f9

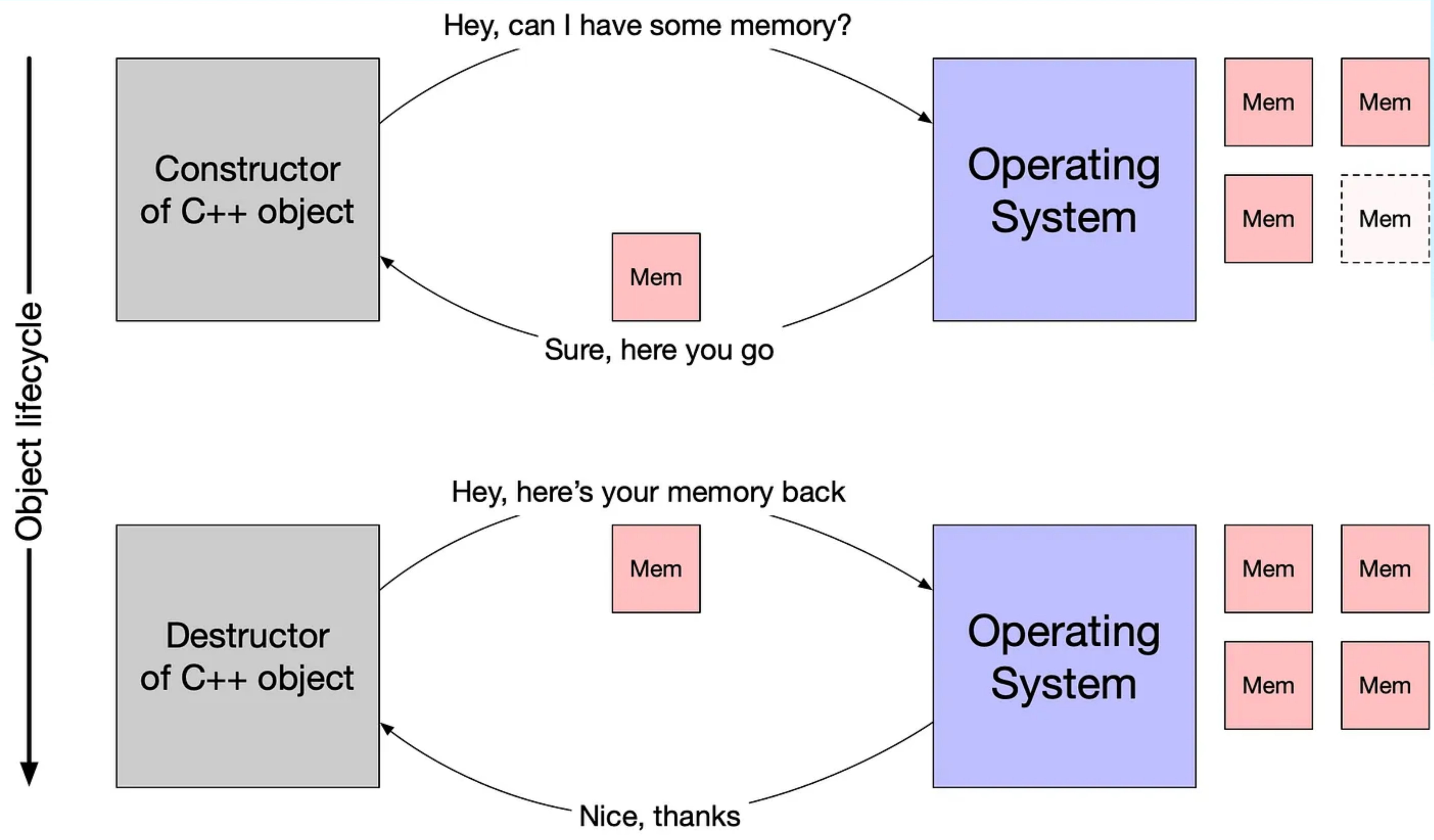
- RAII = **Resource Acquisition Is Initialization**
- This is a *programming idiom* that ensures ensure that resource (=memory) acquisition happens at the same time as the object is initialised
  - All resources needed for the object are created and made ready **in a single line of code**, leading to correct out-of-scope behaviour

- Practically this means that smart pointers (and the like) implementing RAII:
  - Give **ownership** of any **allocated resource** (e.g. dynamically allocated memory) to a (`lvalue`) **object**
  - it is the **destructor** of this object that contains the code to **delete/free the resource** and do the cleanup
  - TLDR: this turns a pointer and its memory management into a class!

MANCHESTER 1824 The University of Manchester

The RAII idiom

https://medium.com/swlh/what-is-raii-e016d00269f9



- A smart pointer is helping this happen "behind the scenes" for the C++ object you create!

Why/when to use smart pointers

- A wrapper class to a pointer is slightly less efficient than a raw pointers
  - but it's **more usable** as there is no chance of memory leaks
  - when you initialise a raw pointer or resource handle to point to an actual resource, you should still **pass the pointer to a smart pointer immediately**

- In modern C++, raw pointers should only used in:
  - small code blocks of limited scope
  - loops
  - helper functions where performance is critical
  - where there is no confusion on who owns the pointer (see later)
  - in your assignments from now on: **use smart pointers, not raw pointers!**

Smart pointers: memory ownership

A nice set of lecture notes: https://github-pages.ucl.ac.uk/research-computing-with-cpp/02cpp1/sec05Pointers.html

- **Memory ownership** is a concept that will come up a lot in smart pointers
  - **Unique ownership:** memory (and data in it) held until needed by a **single variable** (lifetime of data == lifetime of variable)
    - If the variable goes out of scope, the memory is freed
  - **Shared ownership:** memory held until needed by multiple variables (lifetime of data == lifetime of multiple variables)
    - As long as at least one of these variables is in scope, the memory is kept around
  - **Non-owning** (pointers): no connection between lifetime of data/variables and lifetime of memory
    - When non-owning pointer goes out of scope, the memory and data remain
    - This is the behaviour of a raw pointer

Smart pointers: concepts (1)

- unique_ptr
  - Allows exactly **one owner** of the underlying pointer.
  - Replaces the older syntax auto_ptr (now deprecated)
  - **Use this as your default choice**, unless you know for sure that you require a shared_ptr.
  - Can be moved (`move` syntax) to a new owner, but not copied (or passed by value, which makes a copy) or shared
    - this would create confusion on who the owner is, and subsequently on who cleans it up
  - unique_ptr is small and efficient;
    - the size is equivalent to one raw pointer
    - it supports `rvalue` references for fast insertion
    - it can be easily retrieved from STL collections (see lecture 9/10)

MANCHESTER
1824
The University of Manchester

Smart pointers: concepts (2)

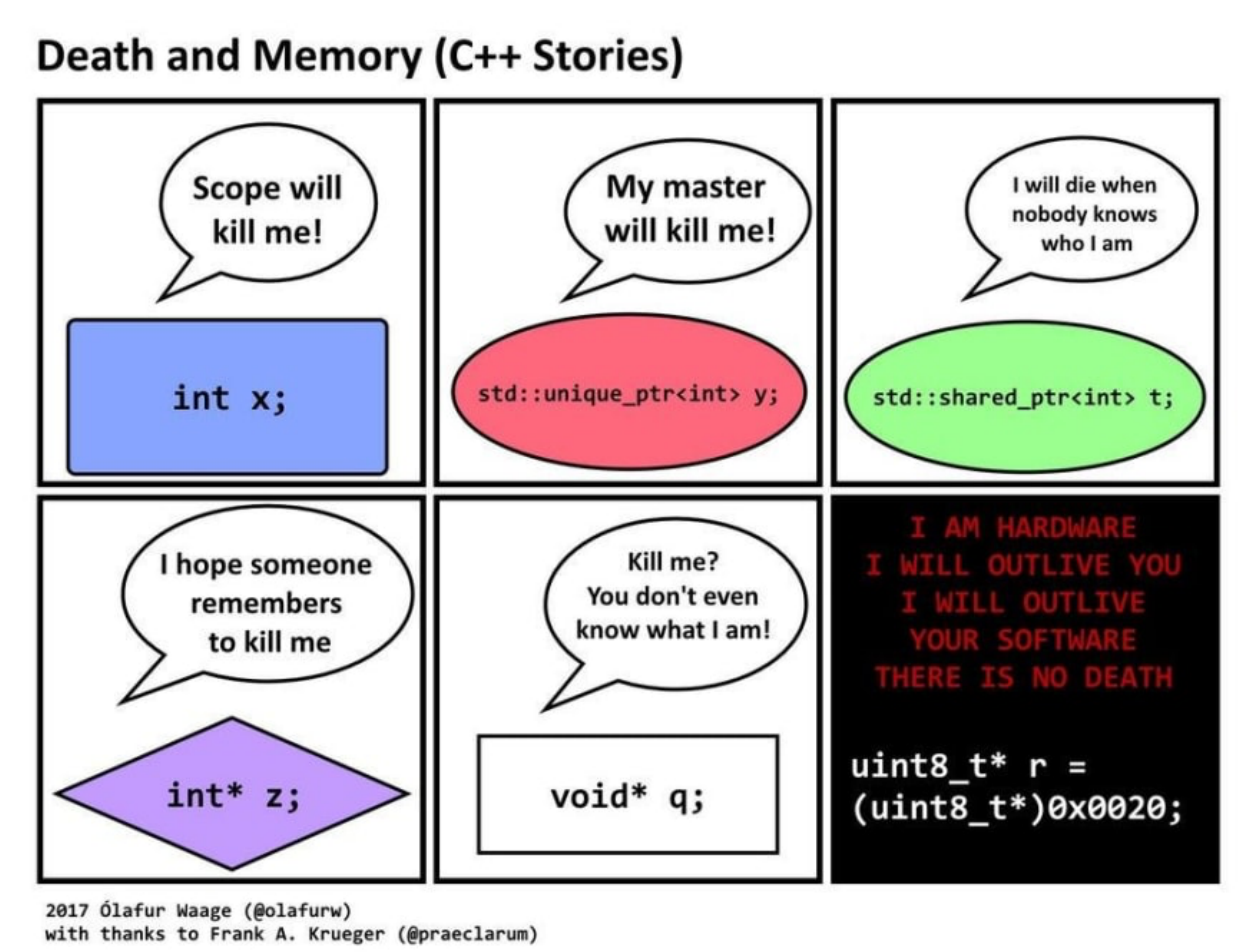
- shared_ptr
    - Reference-counted smart pointer
        - This means: more pointers point to the same location of memory
        - The counter counts how many pointers point to that location of memory
            - When the last one goes out of scope or is made to point to other data, memory is freed
    - Shared pointers can be copied / passed by value
    - Use when you want to assign one raw pointer to **multiple owners**.
    - The raw pointer is not deleted until all shared_ptr owners have gone out of scope or have otherwise given up ownership.

- **weak_ptr (not needed or evaluated, here for the record)**
    - Special-case smart pointer for use in conjunction with shared_ptr.
    - provides access to an object that is owned by one or more shared_ptr instances, but **does not participate in reference counting**.
    - Use when you want to observe an object, but do not require it to remain alive (in scope)
    - Required in some cases to break circular references between shared_ptr instances.

MANCHESTER
1824
The University of Manchester

Difference between unique and shared ptr

https://dev.to/10xlearner/memory-management-and-raii-4f20


Death and Memory (C++ Stories)
Scope will kill me!
int x;
My master will kill me!
std::unique_ptr<int> y;
I will die when nobody knows who I am
std::shared_ptr<int> t;
I hope someone remembers to kill me
int* z;
Kill me? You don't even know what I am!
void* q;
I AM HARDWARE
I WILL OUTLIVE YOU
I WILL OUTLIVE
YOUR SOFTWARE
THERE IS NO DEATH
uint8_t* r =
(uint8_t*)0x0020;
2017 Ólafur Waage (@olafurw)
with thanks to Frank A. Krueger (@praeclarum)

- if you find this really funny, come do some research in my group, we'll have a blast

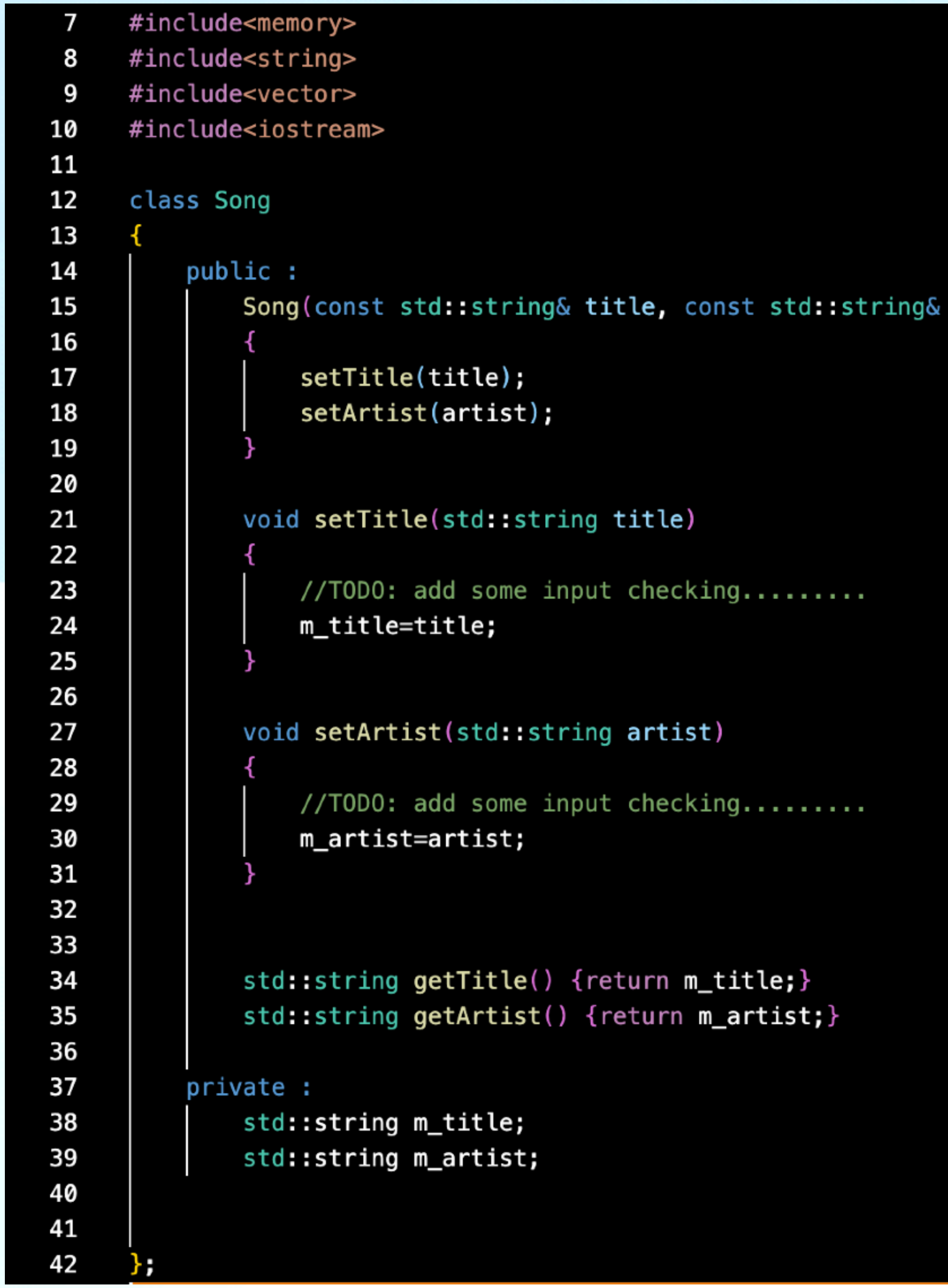
Smart pointers: concepts

- shared_ptr
  - Reference-counted smart pointer
    - This means:
  - Use when you want to assign one raw pointer to **multiple owners**.
  - The raw pointer is not deleted until all shared_ptr owners have gone out of scope or have otherwise given up ownership.

- **weak_ptr**
  - Special-case smart pointer for use in conjunction with shared_ptr.
  - provides access to an object that is owned by one or more shared_ptr instances, but **does not participate in reference counting**.
  - Use when you want to observe an object, but do not require it to remain alive (in scope)
  - Required in some cases to break circular references between shared_ptr instances.

Example of unique_ptr

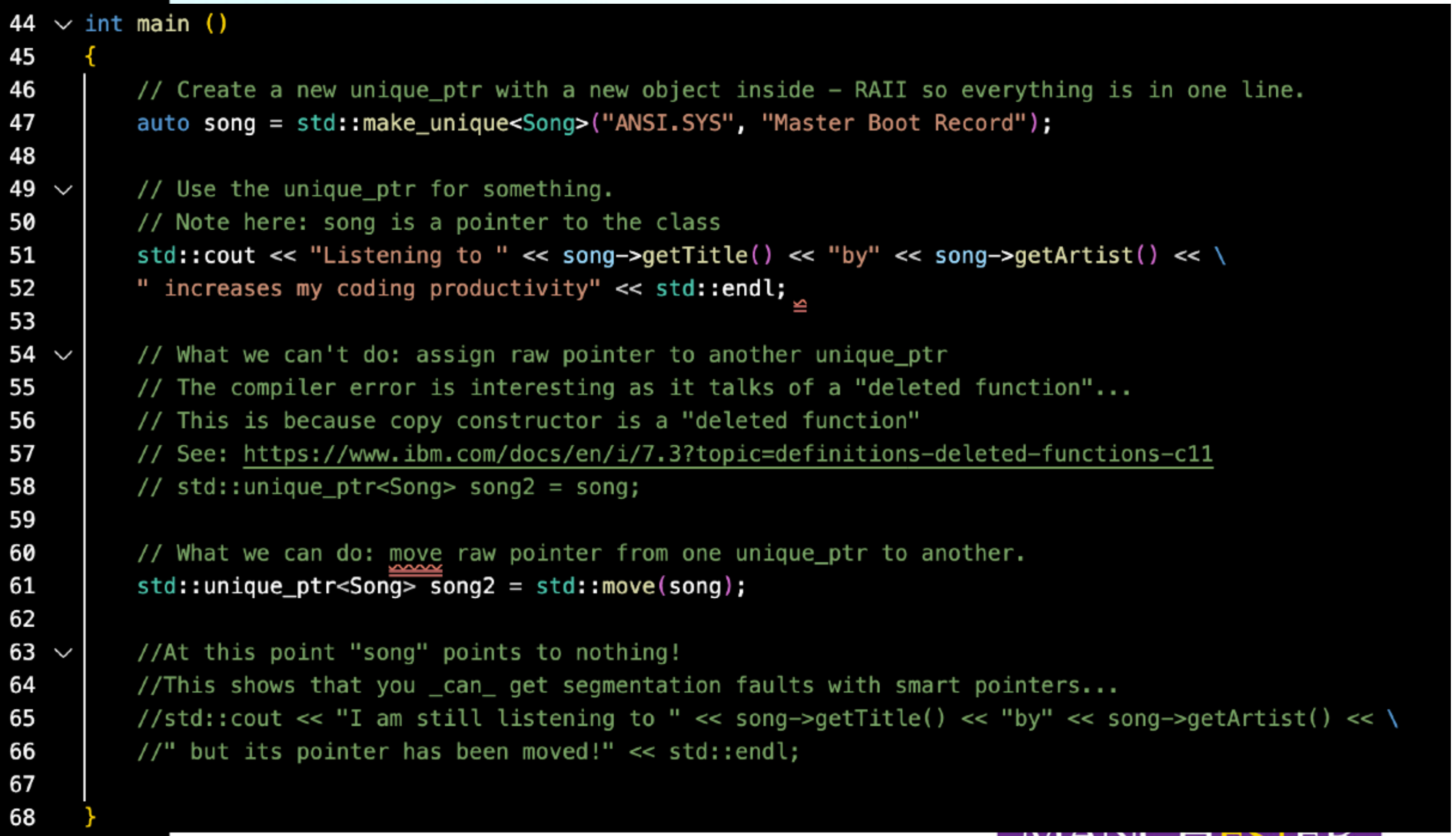
Inspired by: https://learn.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-unique-ptr-instances?view=msvc-170



```cpp
#include<memory>
#include<string>
#include<vector>
#include<iostream>

class Song
{
    public :
        Song(const std::string& title, const std::string& artist)
        {
            setTitle(title);
            setArtist(artist);
        }

        void setTitle(std::string title)
        {
            //TODO: add some input checking.........
            m_title=title;
        }

        void setArtist(std::string artist)
        {
            //TODO: add some input checking.........
            m_artist=artist;
        }


        std::string getTitle() {return m_title;}
        std::string getArtist() {return m_artist;}

    private :
        std::string m_title;
        std::string m_artist;


};
```

Code on GitHub at: Prelecture7/unique_ptr.cpp

- Note also the **use of the keyword auto**
  - Not an auto pointer!!!
  - Since C++11, the compiler deducts the variable type for you! See here
  - Especially useful for loops readability

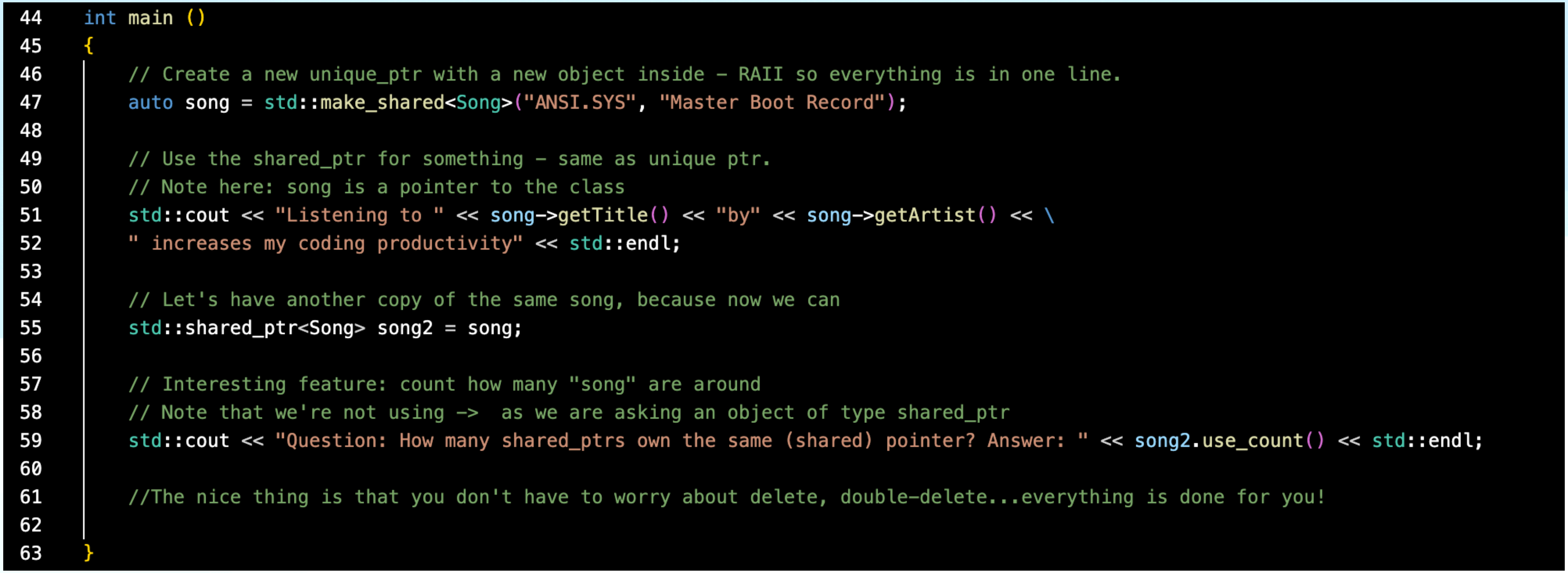


```cpp
int main ()
{
    // Create a new unique_ptr with a new object inside - RAII so everything is in one line.
    auto song = std::make_unique<Song>("ANSI.SYS", "Master Boot Record");

    // Use the unique_ptr for something.
    // Note here: song is a pointer to the class
    std::cout << "Listening to " << song->getTitle() << "by" << song->getArtist() << \
    " increases my coding productivity" << std::endl;

    // What we can't do: assign raw pointer to another unique_ptr
    // The compiler error is interesting as it talks of a "deleted function"...
    // This is because copy constructor is a "deleted function"
    // See: https://www.ibm.com/docs/en/i/7.3?topic=definitions-deleted-functions-c11
    // std::unique_ptr<Song> song2 = song;

    // What we can do: move raw pointer from one unique_ptr to another.
    std::unique_ptr<Song> song2 = std::move(song);

    //At this point "song" points to nothing!
    //This shows that you _can_ get segmentation faults with smart pointers...
    //std::cout << "I am still listening to " << song->getTitle() << "by" << song->getArtist() << \
    //" but its pointer has been moved!" << std::endl;

}
```

Example of shared_ptr

Inspired by previous link & https://en.cppreference.com/w/cpp/memory/shared_ptr/use_count

Code on GitHub at: Prelecture7/shared_ptr.cpp



```cpp
int main ()
{
    // Create a new unique_ptr with a new object inside - RAII so everything is in one line.
    auto song = std::make_shared<Song>("ANSI.SYS", "Master Boot Record");

    // Use the shared_ptr for something - same as unique ptr.
    // Note here: song is a pointer to the class
    std::cout << "Listening to " << song->getTitle() << "by" << song->getArtist() << \
    " increases my coding productivity" << std::endl;

    // Let's have another copy of the same song, because now we can
    std::shared_ptr<Song> song2 = song;

    // Interesting feature: count how many "song" are around
    // Note that we're not using ->  as we are asking an object of type shared_ptr
    std::cout << "Question: How many shared_ptrs own the same (shared) pointer? Answer: " << song2.use_count() << std::endl;

    //The nice thing is that you don't have to worry about delete, double-delete...everything is done for you!

}
```

- The weak_ptr example is left as an exercise to the reader I hate when books do this...see here
    - You won’t need it for your project