C++ Programming I

C++ 11 Smart Pointers

C++ Programming
May 17, 2018

Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- C++ 11 Smart Pointers

Agenda

- C++ 11 Smart Pointers
- `unique_ptr`

Agenda

- C++ 11 Smart Pointers
- `unique_ptr`
- `shared_ptr`

Agenda

- C++ 11 Smart Pointers
- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

Agenda

- C++ 11 Smart Pointers
- `unique_ptr`
- `shared_ptr`
- `weak_ptr`
- Implementing Smart Pointers

Bern University
of Applied Sciences

C++ 11 Smart Pointers

Why Smart Pointers

RAII

```
1  void someFunction()
2  {
3      Resource *ptr = new Resource; // Resource is a class
4
5      // do stuff with ptr here
6
7      delete ptr;
8  }
```


Why Smart Pointers

RAII

```cpp
#include <iostream>

void someFunction()
{
    Resource *ptr = new Resource;

    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        return; // hoppla, the function returns early

    // do stuff with ptr here

    delete ptr;
}
```

Why Smart Pointers

RAII

```cpp
#include <iostream>

void someFunction()
{
    Resource *ptr = new Resource;

    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        return; // hoppla, the function returns early

    // do stuff with ptr here

    delete ptr;
}
```

- **RAII paradigm** - Resource Acquisition is Initialization
- Allocate memory in your constructor
- Deallocate it in your destructor

Why Smart Pointers

RAII

```cpp
#include <iostream>

void someFunction()
{
    Resource *ptr = new Resource;

    int x;
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        return; // hoppla, the function returns early

    // do stuff with ptr here

    delete ptr;
}
```

- **RAII paradigm** - Resource Acquisition is Initialization
- Allocate memory in your constructor
- Deallocate it in your destructor

Smart Pointer Class

We should have a class managing dynamic memory!

Why Smart Pointers

Smart Pointer classes to the rescue?

```
1  #include <iostream>
2
3  template<class T>
4  class auto_ptr
5  {
6      T* m_ptr;
7  public:
8      // Pass in a pointer to "own" via the constructor
9      auto_ptr(T* ptr=nullptr) : m_ptr(ptr)
10     {
11     }
12
13     // The destructor will make sure it gets deallocated
14     ~auto_ptr()
15     {
16         delete m_ptr;
17     }
18
19     // Operators
20     T& operator*() const { return *m_ptr; }
21     T* operator->() const { return m_ptr; }
22 }
```

Why Smart Pointers

Smart Pointer classes to the rescue?

```
1  // A sample class to prove the above works
2  class Resource
3  {
4  public:
5      Resource() { std::cout << "Resource acquired\n"; }
6      ~Resource() { std::cout << "Resource destroyed\n"; }
7  };
8
9  int main()
10 {
11     auto_ptr<Resource> res(new Resource);
12     return 0;
13 }
14
15 // Output
16 Resource acquired
17 Resource destroyed
```

- Note the allocation of memory with `new`
- No explicit delete needed!
- If `res` goes out of scope it's destroyed for us

Smart Pointer

The Resource will be guaranteed to be destroyed, regardless of how the function terminates

Why Smart Pointers

Smart Pointer classes to the rescue?

- Do you see a problem in the next example?

```
1  int main()
2  {
3          Auto_ptr1<Resource> res1(new Resource);
4          Auto_ptr1<Resource> res2(res1); // !
5          return 0;
6  }
```


Why Smart Pointers

Smart Pointer classes to the rescue?

- Do you see a problem in the next example?

```
1  int main()
2  {
3          Auto_ptr1<Resource> res1(new Resource);
4          Auto_ptr1<Resource> res2(res1); // !
5          return 0;
6  }
```

- Resource destroyed twice!

Why Smart Pointers

Smart Pointer classes to the rescue?

- Do you see a problem in the next example?

```
1  int main()
2  {
3          Auto_ptr1<Resource> res1(new Resource);
4          Auto_ptr1<Resource> res2(res1); // !
5          return 0;
6  }
```

- Resource destroyed twice!
- Since C++ 98 until C++ 11, `auto_ptr` was C++ 's first attempt at a standardized smart pointer
- `auto_ptr` implements move semantics through the copy constructor and copy assignment operator, since move semantics was not existing

Why Smart Pointers

Smart Pointer classes to the rescue?

- Do you see a problem in the next example?

```
1  int main()
2  {
3          Auto_ptr1<Resource> res1(new Resource);
4          Auto_ptr1<Resource> res2(res1); //
5          return 0;
6  }
```

- Resource destroyed twice!
- Since C++ 98 until C++ 11, `auto_ptr` was C++ 's first attempt at a standardized smart pointer
- `auto_ptr` implements move semantics through the copy constructor and copy assignment operator, since move semantics was not existing

C++ 11

Since C++ 11 three new smart pointer types are available!

Smart Pointers

Existing Smart Pointers: `#include <memory>`

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- `std::auto_ptr`

Smart Pointers

Existing Smart Pointers: `#include <memory>`

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- ~~`std::auto_ptr`~~ (depriciated since C++ 11, removed in C++ 17)

Smart Pointers

Existing Smart Pointers: `#include <memory>`

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- ~~`std::auto_ptr`~~ (depriciated since C++ 11, removed in C++ 17)
- `std::auto_ptr` is depriciated since C++ 11 because:
  1. it implements move semantics through the copy constructor and assignment operator and, thus, does not behave as expected with other standart library classes
  2. it does not work with dynamically allocated array's (no array deleter)

Garbage Collection

Using Smart pointers is the simplest garbage collector we could think of!

unique_ptr

unique_ptr

Exclusive Ownership

- `unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope.
- Manages a single object (e.g. allocated with `new`)
- Manages a dynamically-allocated array of objects (e.g. allocated with `new[]`)

```
1 // unique pointer to int
2 std::unique_ptr<int> intPtr(new int(42));
3
4 // unique pointer to int array
5 std::unique_ptr<int[]> intPtr(new int[100]);
```

Demo

The usage of a `unique_ptr` is demonstrated in the lecture

shared_ptr

shared_ptr

Shared Ownership


- `std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer.
- Several `shared_ptr` objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:
  1. the last remaining `shared_ptr` owning the object is destroyed
  2. the last remaining `shared_ptr` owning the object is assigned another pointer via operator= or reset()
- A custom deleter is necessary for managing a dynamic array

Demo

The usage of a `shared_ptr` is demonstrated in the lecture

weak_ptr

weak_ptr

Non-owning Pointer

- `std::weak_ptr` is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by `std::shared_ptr`
- It must be converted to `std::shared_ptr` in order to access the referenced object
- `std::weak_ptr` models temporary ownership
- `std::weak_ptr` is used to break circular references of `std::shared_ptr`

Demo

The usage of a `weak_ptr` is demonstrated in the lecture

Implementing Smart Pointers

Implementing Unique Pointers

Exercise 10

- In exercise 10 you are implementing your own smart pointers!
- `unique_ptr`
  - To disable a constructor or operator, *e.g.* the copy constructor, use:

```
1 ClassName(const ClassName&) = delete;
2 ClassName& operator=(const ClassName&) = delete;
```

- `shared_ptr`
  - In addition, implement copy constructor and copy-assignment operator
  - Implement a reference counting mechanism which:
    - Increases the `useCount` when constructed or copied
    - Decreases the `useCount` when destructed or assigned
  - The resource is deleted when the `useCount` is zero

Literature

Read lesson 26 *Understanding Smart Pointers* in the book and search the web!

Thank You

Questions

???