Introductory Robot Programming

ENPM809Y

Lecture 06 – Smart Pointers

Zeid Kootbally

zeidk@umd.edu

Fall 2020

Overview I

Introduction

- Raw pointers are very powerful for memory management on the heap.
  - Allocate.
  - Deallocate.
  - Manage the lifetime of dynamic variables.
- Issues:
  - Uninitialized pointers (wild pointers).
  - Memory leak.
  - Dangling pointers.
- Smart pointers can help reduce all these issues.
  - Ownership relationship.
  - Automatic memory deallocation.

Introduction

- Pointers are part of C++ since the very beginning (we got them from C).
- There was always the tendency in C++ to make the handling with pointers more type-safe without paying the extra cost.
- `std::auto_ptr` was introduced in C++98 to express exclusive ownership. However, `std::auto_ptr` had a major issue.
  - When you copy an `std::auto_ptr` the resource will be moved.
  - What looks like a copy operation was actually a move operation.
- `std::auto_ptr` was extremely bad and the reason for a lot of serious bugs. `std::auto_ptr` was deprecated in C++11 and finally removed in C++17.
- Therefore, we got `std::unique_ptr` with C++11 along with `std::shared_ptr`, and `std::weak_ptr`.

Arrow and Dot Operators

- The `.` (dot) operator and the `->` (arrow) operator are used to reference individual members of `classes`, `structures`, and `unions`.
- The dot operator is applied to the actual object. The arrow operator is used with a pointer to an object.
- Simply saying: To access members of a structure, use the dot operator. To access members of a structure through a pointer, use the arrow operator.

```
struct Person {
    std::string first_name;
    std::string last_name;
};
int main(){
    Person person;
    person.first_name = "Bjarne";
    person.last_name = "Stroutstrup";
    Person *person_ptr {&person};
    person_ptr->first_name = "Guido";
    person_ptr->last_name = "van Rossum";
}
```

Smart Pointers

- An object (from the Standard Library) whose values can be used like pointers, but which provides the additional feature of automatic memory management (requires `#include<memory>`)
- Can only point to heap-allocated memory.
- Implemented as class templates (very similar to vectors).
- Wrapper[1] around a raw pointer.
- Can be dereferenced.

[1] A data structure or software that contains ("wraps around") other data or software, so that the contained elements can exist in the newer system.

Smart Pointers | RAII Principle

- Smart pointers adhere to the Resource Acquisition Is Initialization (RAII) principle.
- The main principle of RAII is to give ownership of any heap-allocated resource to a stack-allocated object whose destructor contains the code to delete or free the resource and also any associated cleanup code.
  - Binds the life cycle of a resource that must be acquired before use (e.g., allocated heap memory) to the lifetime of an object (e.g., smart pointer).
  - RAII guarantees that the resource is available to any function that may access the object.
  - RAII also guarantees that all resources are released when the lifetime of their controlling object ends.

Smart Pointers | RAII Principle

- The following example compares a raw pointer declaration to a smart pointer declaration.

```
void UseRawPointer(){
    //--using a raw pointer -- not recommended.
    int *ptr {new int{3}};
    //--use ptr in the body
    //--don't forget to delete!
    delete ptr;
}


void UseSmartPointer(){
    //--declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<int> ptr = std::make_unique<int>(3);
    //--use ptr in the body
} //--ptr is deleted automatically here.
```

Smart Pointers | Smart Vs. Raw Pointers

- When to use raw pointers?
  - Mostly in code that is oblivious to memory ownership.
  - Typically in functions which get a pointer from some place else (parameter of the function) and do not allocate nor deallocate, and do not store a copy of the pointer which outlasts their execution.

```cpp
void Print(int *ptr){
    std::cout << *ptr << std::endl;
}
```

- When to use smart pointers?
  - Any time you need to allocate memory on the heap.

Smart Pointers | unique_ptr, shared_ptr, and weak_ptr

- Three different types of smart pointers for different purposes. The type of smart pointer used clearly shows your intention in the code.
  - **`std::unique_ptr`** is a scope pointer. Use `std::unique_ptr` when you do not intend to hold multiple references to the same object. For example, use it for a pointer to memory which gets allocated on entering some scope and deallocated on exiting the scope.
  - Use **`std::shared_ptr`** when you do want to refer to your object from multiple places - and do not want your object to be deallocated until all these references are themselves gone.
  - Use **`std::weak_ptr`** when you do want to refer to your object from multiple places with no ownership involved.

Smart Pointers | unique_ptr

```
std::unique_ptr<T> ptr {new T{value}};
```

or

```
std::unique_ptr<T> ptr = std::make_unique<T>(value);
```

- `std::make_unique` (since C++14) is a helper function and provides a better way to initialize `std::unique_ptr`.
- `make_unique` returns a `unique_ptr` of a specified type and allows us to pass initialization values into the constructor for the managed object.

- Points to an object of type `T` on the heap and owns `T`.
- There can only be at most one `unique_ptr<T>` pointing to `T`.
- The pointer cannot be copied or assigned.
- It can be moved with `std::move()`.
- When the pointer is destroyed, what it points to is automatically deallocated (no need to use `delete`).

Smart Pointers | unique_ptr

```cpp
std::unique_ptr<int> ptr {new int{3}};
```

- A smart pointer is a class template that you declare on the stack and initialize by using a raw pointer that points to a heap-allocated object.
- After the smart pointer is initialized, it owns the raw pointer.
- This means that the smart pointer is responsible for deleting the memory that the raw pointer specifies.
- The smart pointer destructor contains the call to `delete`, and because the smart pointer is declared on the stack, its destructor is invoked when the smart pointer goes out of scope.

Smart Pointers | unique_ptr

```cpp
int main() {
    {
        //--p1 points to 100 on the heap
        std::unique_ptr<int> p1{new int{100}};
        std::cout << *p1 << std::endl;//--100
        *p1 = 200;
        std::cout << *p1 << std::endl;//--200
    }//--automatic deallocation of memory for p1
    std::unique_ptr<int> p2 = std::make_unique<int>(30);
    std::cout << *p2 << std::endl;//--30
}//--automatic deallocation of memory for p2
```

Smart Pointers | unique_ptr

- `std::unique_ptr` comes with a set of methods.
- `reset()` can be used to replace the managed object (deallocate memory and allocate memory for new object on the heap).
- `get()` returns a raw pointer to the managed object or `nullptr` if no object is owned.

```
int main(){
    //--p1 points to 100 on the heap
    std::unique_ptr<int> p1{new int{100}};
    std::cout << p1.get() << std::endl;//--0x55833f2b2e70
    p1.reset(nullptr);//--p1 is now nullptr
    if (p1)//-- if p1 != NULL
        std::cout << *p1 << std::endl;//--will not execute
}
```

- ➤ CLion:Lecture06:main.cpp

Smart Pointers | unique_ptr | Move Semantics

- Move semantics allows an object, under certain conditions, to take ownership of some other object's external resources. This is important in two ways:
  - Turning expensive copies into cheap moves.
  - Implementing safe "move-only" types; that is, types for which copying does not make sense, but moving does. Examples include locks, file handles, and smart pointers with unique ownership semantics.
- `std::move` is used to indicate that an object may be "moved from", i.e., allowing the efficient transfer of resources from an object to another object.
- After moving a `std::unique_ptr` you cannot use it because you transferred ownership (just like when you sell a car).

Smart Pointers | unique_ptr | Move Semantics

```cpp
int main(){
    std::unique_ptr<int> p1 = std::make_unique<int>(100);
    std::cout << p1.get() << std::endl;//--0x562888ef6e70
    std::unique_ptr<int> p2{std::move(p1)};
    std::cout << p1.get() << std::endl;//--point to nullptr
    std::cout << p2.get() << std::endl;//--0x562888ef6e70
}
```

- ➤ CLion:Lecture06:main.cpp

Smart Pointers | shared_ptr

- The `shared_ptr` type is a smart pointer in the C++ Standard Library that is designed for scenarios in which more than one owner might have to manage the lifetime of the object in memory.
- After you initialize a `shared_ptr` you can copy it, pass it by value in function arguments, and assign it to other `shared_ptr` instances.
- All the instances point to the same object, and share access to one "control block" that increments and decrements the reference count whenever a new `shared_ptr` is added, goes out of scope, or is reset.
- When the reference count reaches zero, the control block deletes the memory resource and itself.

Smart Pointers | shared_ptr

```cpp
std::shared_ptr<T> ptr {new T{value}};
```

or

```cpp
std::shared_ptr<T> ptr = std::make_shared<T>(value);
```

- Points to an object of type `T` on the heap and owns `T` (similar to `std::unique_ptr`).
- There can be many `shared_ptr<T>` pointing to `T`.
- Establishes shared ownership relationships.
- The pointer can be copied or assigned.
- It can be moved with `std::move()`.

Smart Pointers | shared_ptr | use_count()

- We can have multiple `std::shared_pointer` referencing the same object on the heap.
- How does C++ know when that object needs to be destroyed?
    - The most common techniques is to use a reference count.
    - Each time we instantiate a `std::shared_pointer` object and have it point to or reference a heap object we increment a counter.
    - This counter has the number of `std::shared_pointer` that reference(s) the heap object.
    - When the reference count is 0, then the heap object is destroyed.
    - If `use_count` returns 0, the shared pointer is empty and manages no objects (whether or not its stored pointer is null).
    - If `use_count` returns 1, there are no other owners.

Smart Pointers | shared_ptr | use_count()

```cpp
int main(){
    {
        //--p1 points to 100 on the heap
        std::shared_ptr<int> p1 = std::make_shared<int>(100);
        std::cout << *p1 << std::endl;//--100
        *p1 = 200;
        std::cout << *p1 << std::endl;//--200
    }//--automatic deallocation of memory
}
```

Smart Pointers | shared_ptr | use_count()

```cpp
int main(){
    //--p1 points to 100 on the heap
    std::shared_ptr<int> p1 = std::make_shared<int>(100);
    std::cout << p1.use_count() << std::endl;//--1

    std::shared_ptr<int> p2{p1};//--shared ownership
    std::cout << p1.use_count() << std::endl;//--2
    std::cout << p2.use_count() << std::endl;//--2

    p1.reset();//--decrement the use_count, p1 is nullptr
    std::cout << p1.use_count() << std::endl;//--0
    std::cout << p2.use_count() << std::endl;//--1
}
```

- ➤ CLion:Lecture06:main.cpp

Smart Pointers | weak_ptr

- `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: when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else, `std::weak_ptr` is used to track the object, and it is converted to `std::shared_ptr` to assume temporary ownership. If the original `std::shared_ptr` is destroyed at this time, the object's lifetime is extended until the temporary `std::shared_ptr` is destroyed as well.
- Another use for `std::weak_ptr` is to break reference cycles formed by objects managed by `std::shared_ptr`. If such cycle is orphaned (i,e. there are no outside shared pointers into the cycle), the `std::shared_ptr` reference counts cannot reach zero and the memory is leaked. To prevent this, one of the pointers in the cycle can be made weak.
- ➤ CLion:Lecture06:main.cpp

Next Class | 10/15

- Lecture07: Object Oriented Programming – Part I.
- Quiz on raw and smart pointers.
- Assignment due.
- Stay safe!