CS 246 Fall 2018 — Tutorial 11

November 28, 2018

Summary

1 Resource Acquisition is Initialization (RAII)

- Consider the following code segments:

```
// Code segment 1
try{
    int *arr1 = new int[10];
    int *arr2 = new int[20];
} catch (std::bad_alloc) {
    // Can you make sure there is no
    // memory leak when the exception
    // is caught? Why not?
}
```

```
// Ugly "Fix" to code segment 1
try{
    int *arr1 = new int[10];
    int *arr2 = nullptr;
    try {
        arr2 = new int [20];
    } catch (std::bad_alloc &e) {
        delete arr1;
        throw;
    }
} catch (std::bad_alloc &e) {
}
```

- How can we ensure that heap allocated memory is freed properly, when taking exception handling into account?
- Idea: wrap all memory allocation (**resource acquisition**[1]) into constructors (**initalization**)!
- This practice is generally referred to **Resource Acquisition Is Initialization (RAII)**.
- *RAII* is vital to writing exception-safe code in C++.
- RAII relies on the C++ guarantee that when an exception is thrown, stack-allocated memory will be reclaimed.
  - In particular, **destructors for stack-allocated objects will run**.

[1] There are more types of resource acquisition. e.g. opening a file, opening a socket, or acquiring a lock.

- Under RAII, Resources are acquired using stack-allocated object initialization (i.e. through its constructor), so that the resource cannot be used before they are available and are "released" when the owning object is destroyed.
- The code segment above could be written as this, using RAII:

```
struct Wrapper{
    int *arr = nullptr;
    int length;
    Wrapper(int length):
      length{length}{
        arr = new int[length];
    }
    ~Wrapper(){
        delete arr;
    }
};
```

```
try{
    Wrapper w1{10},w2{20};
    ....
} // Memory taken be Wrapper freed here
catch (std::bad_alloc e){
    ....
}
```

- Making use of RAII also more easily facilitates implementing the various levels of exception safety.

2 Levels of Exception Safety

- While we have established that RAII is vital to writing exception-safe code, it would be ideal to be able to tell someone how safe the code. There are three levels of exception safety. Each describes to what can be expected of code if an exception is thrown.
  1. **Basic** guarantee: if an exception is thrown, data will be in a valid state and all class invariants are maintained.
     - Example: If we change variables in an assignment operator before allocating heap memory with `new`.
  2. **Strong** guarantee: if an exception is thrown, the data will appear as if nothing happened.
     - Example: The copy-and-swap idiom for the assignment operator provides strong guarantee.
  3. **No-throw** guarantee: an exception is never thrown and the function must always succeed.
     - Example: Swapping two pointers using `std::swap` is guaranteed not to throw an exception.
- Note that if a piece of code matches none of those levels above, the code is said to have **no guarantee**.

3 Smart Pointers

- Dynamic memory pose a problem when trying to implement exception safety in particular.
- The pointer itself is reclaimed but the memory that it points to is not.
  - This could possibly be a very large object on the **heap**.
  - If heap memory is not deleted in a `catch` block, then if an exception occurs, the memory will be leaked.
- The solution to this problem is to follow the RAII idiom, which we have just discussed above.
- However, the wrapper class solution is somewhat complicated; we do not want to explicitly put all allocation in a class, for this leads to excessive class definitions.
- There are wrapper classes provided in STL for pointers pointing to dynamic memory: `unique_ptr`, `shared_ptr`.
  - `unique_ptr` means the only pointer that points to a block of heap memory.
    - `unique_ptr`s are usually used to model composition relationship.
  - `shared_ptr` allows many pointers that all point to the same block of heap memory and only deletes that memory when no other `shared_ptr`s point to it. (Example: `tut11/shared_pointer/`)
  - `shared_ptr`s should only be used if the pointers are all sharing ownership; you should use `unique_ptr` when there is a clear owner (in this case, use raw pointer for "has-a" relationship).
  - Raw pointers still have some uses even if you use smart pointers to manege dynamically allocated memory.

```
// A node for doubly linked list
templete <typename T>
struct Node{
    T data;
    std::unique_ptr<Node<T>> next;
    // Raw pointers are okay to use for modeling "has-a" relationship.
    Node<T> *prev;
};
```

4 Returning unique_ptr

- Can a unique_ptr be copied? Let's try the following code:

```
#include <memory>
using namespace std;
```

```
int  main(){
    unique_pointer<int> n = make_unique<int>(10);
    unique_pointer<int> m = n;
}
```

- What is the expected behaviour? Should m steal the data within n? Should m make it's own copy of n's data?
- Neither! Both the copy constructor and copy assignment operator are disabled. The code for unique_ptr would look something like this:

```
template<T> class unique_ptr<T>{
    T* data = nullptr;
  public:
    unique_ptr() {}
    unique_ptr(T* t): data{t} {};
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr(unique_ptr&& p): data{p.data} { p.data = nullptr; };
    unique_ptr& operator=(const unique_ptr&) = delete;
    unique_ptr& operator=(unique_ptr&& p){ swap(data, t.data); }
    ~unique_ptr(){ delete data; }
    T& operator() { return *data; }
};
```

- This implementation ensures that there will only be one unique_ptr pointing at data meaning data will only be deleted once.
- However, why can we return unique_ptrs by value from functions? Example: unique.cc
- We know that when we return by value a constructor is called. When returning a unique pointer, the move constructor is called (or elision occurs).
- Thinking about it, the function owns the unique_ptr until it goes out of scope and the object it is pointing at should be deleted. When we return a unique_ptr (or any other type) from a function, the ownership of the pointer is being transferred with the returned pointer. Thus, it makes sense to be able to return unique_ptr while also not being able to copy them.