CS 246 Spring 2017 - Tutorial 11

July 23, 2019

1 Summary

- RAII
- Smart Pointers
- Dangers of Smart Pointers
- Exception Safety

2 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 e;
    }
}
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**)! The wrapper will be on the stack.
- This practice is generally referred to **Resource Acquisition Is Initialization (RAII)**.

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

- *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**.
- 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 arr1 = nullptr;
    int arr2 = nullptr;
    Wrapper(){
        arr1 = new int[10];
        arr2 = new int[20];
    }
    ~Wrapper(){
        delete [] arr1;
        delete [] arr2;
    }
};
```

```
try{
    Wrapper w;
    ....
} // Memory taken be Wrapper freed here
catch (std::bad_alloc e){
    ....
}
```

3 Smart Pointers

- In C++11, There are wrapper classes provided in STL for pointers pointing to dynamic memory: `shared_ptr`, `unique_ptr`, and `weak_ptr`[2].
  - `unique_ptr` means the only pointer that points to a block of heap memory.
    - **Note:** `unique_ptr` supports `operator[]` and can support array initalization.
      `unique_ptr<int[]> p{new int[28]};`
    - `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/smart_pointer/group`)
  - `weak_ptr` is similar to `shared_ptr` but doesn't count towards the "shared count".
    - It is used to implement temporary ownership of a `shared_ptr`.
    - It is also used to prevent cyclic ownership in `shared_ptr`s.
    - The usage of `weak_ptr`s is different than `shared_ptr`s and `unique_ptr`s. You should seek external documentation for the proper usage.
      (Example: `tut11/smart_pointer/cache`)

[2]Not discussed in class and not required knowledge.

– Raw pointers still have some uses even if you use smart pointers to manage 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 Dangers of Shared Pointers

- We've seen how smart pointers could make our code "smarter"[3].
- However, there are some things we can do with smart (shared) pointers that are not so smart.

1. If you used smart pointers to manage heap-allocated objects, you should use `std::make_shared` and `std::make_unique` rather than `new` whenever possible.
   - If you maintain both regular pointers (e.g. `int *p;`) and smart pointers to the same piece of heap memory, then you might not realize when the memory is reclaimed. (Example: `tut11/smartptr_dangers/mixed1.cc`)
   - Using raw pointers to create 2 smart pointers have **independent** "count" values, and most certainly will lead to double free errors. (Example: `tut11/smartptr_dangers/mixed2.cc`)
2. You should be careful not to have a cycle of `shared_ptr` or the objects will never get deleted (Example: `tut11/smartptr_dangers/cycle.cc`).

5 Exception Safety

- Making use of RAII also more easily facilitates implementing the various levels of exception safety.
- There are three levels of guarantees that you can provide with respect to exception safety:

1. **Basic** guarantee: if an exception is thrown, data will be in a valid state but may not make sense.
   – 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] Pun halfway intended

3. **No-throw** guarantee: an exception is never thrown.
   - 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**.
- 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.
- Example: `tut11/exception_safety`