CS 246 Fall 2021 — Tutorial 9

November 10, 2021

Summary

1 Exception Handling

- Instead of using C-style strategies of handling errors, we use *exceptions* in C++ to control the behaviour of the program when errors arise.
- If an exception is *raised/thrown*, the program will terminate if there is no matching handler (that is, a `catch` block) for it.
- Recall that we catch exceptions with `try` and `catch` blocks:

```
try {
    throw 42;
} catch (...) { // "..." will accept any type of error
    cerr << "Caught something" << endl;
}
```

- The `try` and `catch` blocks must be used together.
- You can also nest a `try` block inside of another.
- We can throw *anything* in C++, including exception objects, strings, integers, etc.
- We can also `throw` what we just caught in the `catch` block:

```
try {
     throw 42;
} catch (...) {
    throw; // will throw the same exception that was caught
}
```

1.1 Exception and Inheritance

Consider the code below:

```
// One exception class
class BadInput{};

// Another exception class
class BadNumber: public BadInput{
  public:
    string what() { return "no number given"; }
};

int main(){
    try { // Let's try this block:
        int x;
        cout << "Give me a number" << endl;
        cin >> x;
        if (x < 50) throw BadNumber{};
        // Which catch block will run?
    } catch(BadInput &b) {
        cerr << "This is caught" << endl;
    } catch(BadNumber &b) {
        // Accessing auxilliary information from caught object
        cerr << b.what() << " is caught" << endl;
    }
}
```

Question: What will be the output?

Good practice:

- Throw by value, and catch by reference.
- `catch` blocks needs to be ordered from the most specific to the least specific.
  - For example, if there is an inheritance relationship between the exception classes, you should always catch the subclass exceptions first.
  - This implies that the `catch(...)` clause must always be the last `catch` statement.

2 Resource Acquisition is Initialization (RAII)

- Consider the following code segments:

```
// Code example 1
try {
    int *arr1 = new int[10];
    int *arr2 = new int[20]; // what if operator new throws?
    delete [] arr1;
    delete [] arr2;
} catch( std::bad_alloc & e ) {
    // handler
}
```

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

- 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 as *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 raised, 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 it is available and is "released" when the owning object is destroyed.
- The previous code example 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 by Wrapper freed here
catch (std::bad_alloc & e) {
    ...
}
```

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

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

3 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 is. There are three levels of exception safety. Each describes to what can be expected of code if an exception is raised.
  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.
- If a piece of code matches none of those levels above, the code is said to have *no guarantee*.
- **Note:** In this course, we expect the *basic* guarantee as the minimum standard that any function you write should meet.

4 Smart Pointers

- Dynamic memory poses 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`, and `shared_ptr`.

- `unique_ptr` means the only pointer that points to a block of heap memory.
  - `unique_ptr`s are usually used to model the composition relationship.
- `shared_ptr` allows many pointers to all point to the same block of heap memory and only deletes that memory when no other `shared_ptr`s point to it. (Example: `tut09/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 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;
};
```

5 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 its 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_ptr`s 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.