C++

April 3, 2022

Mike Spertus

spertus@uchicago.edu

MASTERS PROGRAM IN COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO

MORE ON WRITING EXCEPTION-SAFE CODE

Exceptions have a pervasive impact on your code

- As we discussed last week, exceptions can cause clean-up code to be missed
  - We learned how to fix this with RAII
    - ```
      void f() {
        auto x = make_unique<X>("foo");
        x->f(g());
      }
      ```
    - `x` will be cleaned up even if `f` or `g` throw an exception
- However, there are a number of other impacts that exceptions can have that you need to be aware of
- Let’s look at a few
      ```

Exceptions and destructors

- Recall that an object's destructor is always called at the end of its lifetime
- That means destructors of automatic duration objects are called during exception processing even though normal code is bypassed
- This is what makes RAII work!
- But..

What if a destructor throws during exception processing?

- Consider the following
-
```
struct X { ~X() {throw "X";} };
void f() {
 auto x = make_unique<X>("foo");
  x->f(g());
}
```
- If `g` throws an exception, the `x`’s destructor will call `X`’s destructor, which will throw an exception in the middle of processing `g`’s exception
- This makes no sense, so the language runtime will immediately call `std::terminate()`, abruptly halting your program
- Oops

Best practice: Don’t throw exceptions from destructors

- To avoid this, don’t throw exceptions in your destructor
- If your destructor calls something that may throw, be sure to catch before returning
-
```
struct X {
  ~X() {
    try {
      g(*this); // g may throw
    } catch(...) {} // Ignoring may be only option
  }
};
```

Writing exception-safe interfaces

- When you write a function or method, you should think about what happens if an exception occurs while passing arguments or return values
- This only applies to pass-by-value arguments
- Let’s look at a real-world example

Designing a stack interface

- In your HW, you were asked to write a stack based on a push to push a value and a pop to get the value back
  - As you are probably familiar with from stacks in other languages
- You may be surprised to see that `std::stack` uses a different interface where `stack::pop()` returns a void (discarding the top element)
- If you want to see the top element before you pop it into oblivion, `stack::top()` gives you a reference to it, so you can copy or move it to your preferred destination

Why does std::stack work this way? Exception-safety!

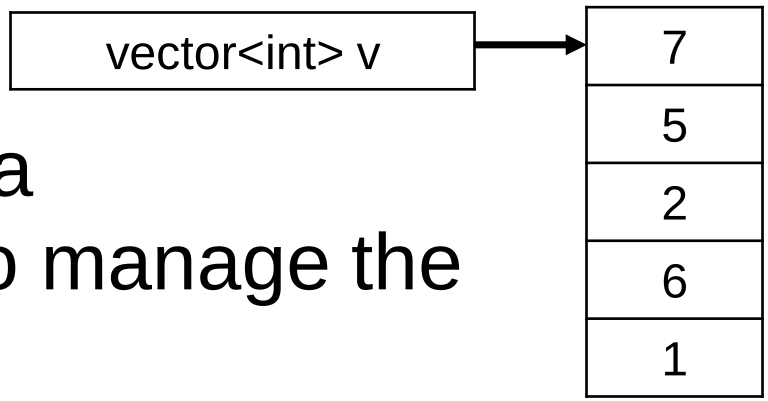
- Again, you would expect std::stack to be able to pop an object of a stack and return its value
  - ```
    stack<A> stk;
    ...
    A a(stk.pop()); // Illegal!
    ```
- The problem with this would be if A's copy or move constructor threw an exception.
  - The top element could be lost forever
- Instead, stack::pop has void return type.
- Do the following instead
  - ```
    stack<A> stk;
    ...
    A a(stk.top());
    stk.pop();
    ```

Can you really program if an exception can be thrown at any time?

- As we just saw, `std::stack` has a convoluted interface imposed on it by exception safety
  - In particular, this interface is bad in multi-threaded programs (why?)
- Here’s an even worse example where not knowing if an exception can be thrown forces inefficient algorithms to be used

How do we grow a vector?

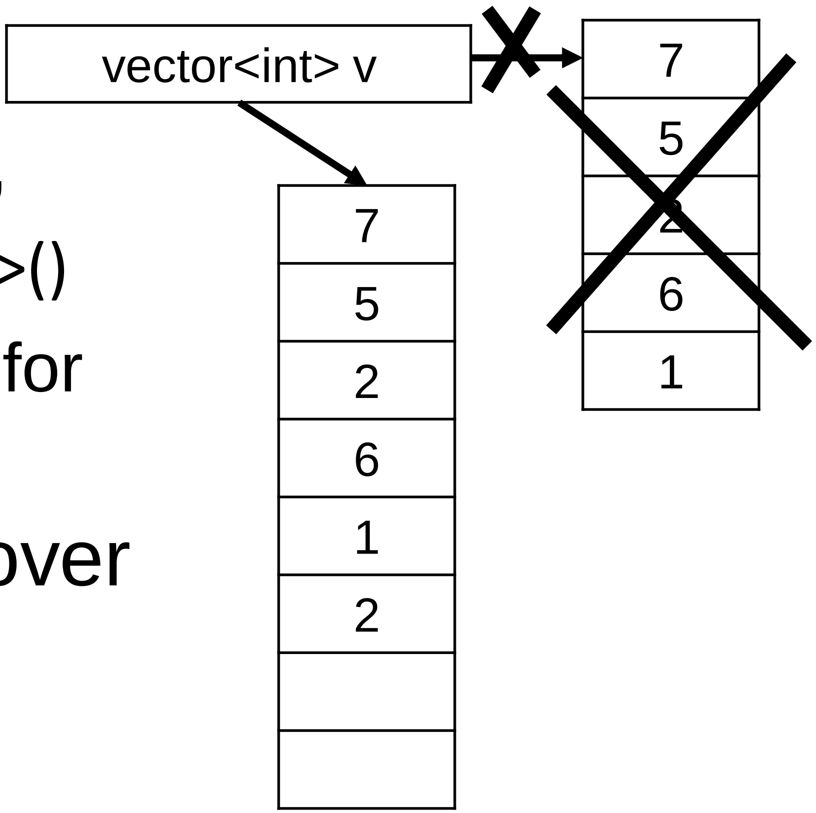
- A vector stores a bunch of objects in a contiguous array it has reserved in memory



- (Logically) it uses a unique_ptr<int[]> to manage the array

How do we grow a vector?

- Suppose we call `push_back(2)` and there is no more room in the array
- vector will reserve a larger array. E.g,
  - `make_unique<int[8]>()`
  - I gave a little extra for future growth
- and copy the `ints` over

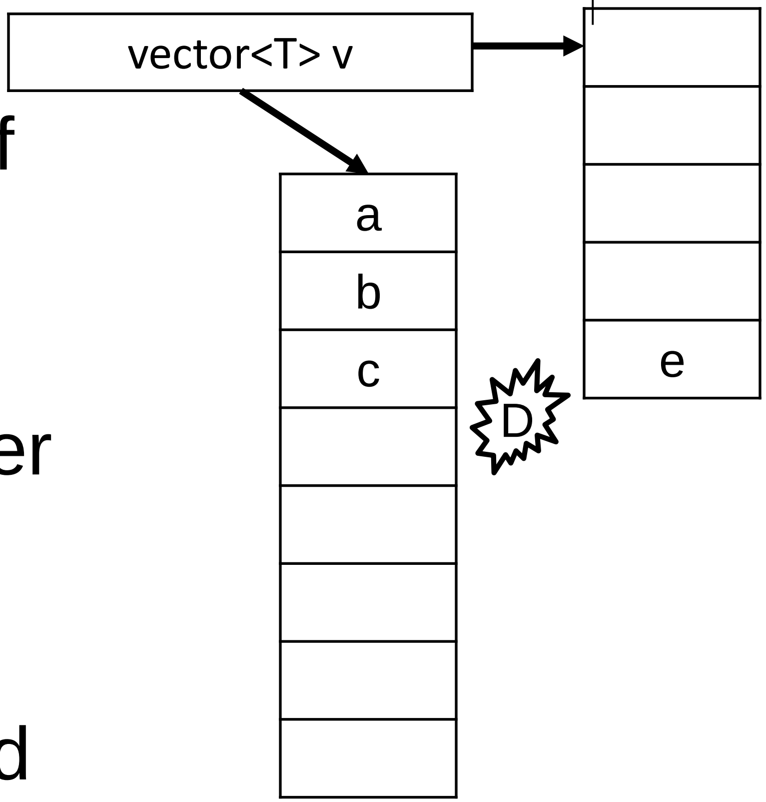


Can we move?

- In the example of a vector<int>, there is no need to move, but what if we have a vector of things that
  - Can’t be copied
    - vector<unique_ptr<int>>
  - Or are expensive to copy but cheap to move
    - vector<HTMLPage>
- We’d rather vector move the objects to their new location

Exceptions rear their ugly head

- What if an exception is thrown while moving one of the objects?
- push_back will throw an exception, but it won't be recoverable because neither the old array or the new array is usable
- In fact, if push_back throws an exception, it is supposed to leave the vector untouched



If there is any chance a move throws, then we have to copy

- In the previous slide, we would have been better off copying than moving
- But performance may be worse
- And something like `vector<unique_ptr<T>>` would simply be impossible
- This is, of course, unacceptable

noexcept

- All of these are symptoms of the fact that writing code that can tolerate an exception at any conceivable moment is a drag at best and disastrous at worse
- While C++ does not have exception specifications like Java, you can label a function or method as `noexcept`, which means it will never throw an exception
  - If it tries, `std::terminate` will be called

How vector decides whether to move or copy

- If the type stored in it has a noexcept move constructor, then it moves it
- If not, it copies
- Since unique_ptr's move constructor is annotated as noexcept
  - unique_ptr(unique_ptr&& u) noexcept;
- To make this easier, there is a type trait is_nothrow_move_constructible_v that Concepts or SFINAE can leverage to call the right code

Wait, I thought C++ had exception specifications

- They used to
  - You would be able to list what exceptions a function could throw
- While the concept might be workable
  - Java has made heavy use of exception specifications for years
- The design was entirely broken, and it was never a good idea to use them
  - http://www.gotw.ca/publications/mill22.htm
- So they were deprecated in C++11 and removed in C++17
  - Given C++' emphasis on backwards compatibility, that should tell you everything you need to know about how terrible they were

Templates and noexcept

- Is our square template noexcept?
  - ```
    template<typename T>
    auto square(T &&x) {
      return x*x;
    }
    ```
- It depends on whether T’s move constructor is noexcept
- You can give noexcept a true or false argument to say so
- ```
  template<typename T>
  auto square(T &&x)
    noexcept(is_nothrow_move_constructible_v<T>) {
    return x*x;
  }
  ```

noexcept(noexcept(...))

- Sometimes you don't have a convenient type trait
- There is also a "function" named noexcept that takes an expression and returns true if it is noexcept
-
```
template<typename T>
auto square(T &&x)
  noexcept(noexcept(x*x)
        && is_nothrow_move_constructible_v<T>) {
   return x*x;
}
```
- This is overkill for square but sometimes you need it
- The keyword noexcept is used to mean a lot of things ☹
- We do this because introducing a new keyword can break existing code

noexcept and destructors

- As we mentioned, your destructors should almost always be noexcept
- In fact, whether you say so or not, your class destructors are implicitly noexcept
- If you really mean for a destructor to be able throw an exception, you’ll have to explicitly say noexcept(false)

Best practice

- If your class' move and copy constructors are noexcept, be sure to declare them that way
  - That will pay off every time you put them in a container
  - Passing by value will likely be safer and more efficient as well
- Don't try to declare everything with the correct noexcept specifier
  - That way madness lies
  - But don't hesitate it if there is a specific benefit

There is a rethinking of error best practices

- While exceptions have been the norm
- We have seen that they are very complicated
- Many important systems (e.g., computer games) have told their compiler to disable exceptions for performance or other reasons
  - The rest of the lecture will dig into this
- This of course means they are not programming in C++
  - It also breaks much of the standard library
    - Game companies write their own exception-free versions of much of the standard library
- But they do it anyway!

`std::nothrow`

- Actually, there is one standards-compliant approach to suppressing exceptions
- At least in the case of memory allocation

`std::nothrow`

- Normally, allocation can throw a `std::bad_alloc` exception
  - auto a = make_unique<T>(); // May throw
  - auto b = new T; // May throw
- `nothrow` allocation returns `nullptr` if allocation fails
  - `auto b = new(nothrow) T; // Will not throw`
- No equivalent for make_unique
  - But see homework

How does `nothrow` allocation work

- Overloading `operator new`!
- Let's review

`operator new`

- `operator new` is similar to `malloc` in C (in fact, it usually just calls `malloc`) in that it allocates the requested number of bytes of memory
- For example, "`operator new(10)`" allocates 10 bytes of memory and returns a `void *` pointing to it
- If the allocation fails (usually because there is not enough memory) it throws a `std::bad_alloc` exception

Can I use operator overloading?

- Yes! You can use C++ operator overloading to define your own `operator new`
  - You can redefine the one taking a `size_t` or you can create your own signatures
- In fact, the standard library defines a couple of useful overloads

Placement new

- Suppose you already have memory (e.g. a buffer) and you want to put an object in it
- In this case, you don’t want to say “new A” because it will put the object someplace else
- Fortunately, the standard library provides an overload that you can tell where you want it to put the object

```
void *operator new(size_t  s, void *p)
  { return p; }
```

- `new(p) A; // The A object will be at location p`

Nothrow new

- Sometimes you don't want new to throw an exception when allocation fails but return `nullptr` instead, similarly to C
  - Low latency applications like gaming and high-speed trading often want this
- The standard library provides a non throwing overload
  - `new(std::nothrow) A; // Won't throw`
- If you want this to apply throughout you entire program just overload the regular operator new to call that

Reminder: You should rarely use `new`

- Use `make_unique` instead
- The reason is exception safety!
- Bringing us back full circle
- Let's review

Why avoid `new`?

- The problem is that new returns an owning raw pointer which violates exception safety by not using RAII:

```
void f()
{
  // g(A *, A *) is responsible for deleting
  g(new A(), new A());
}
```

  - What if the second time A’s constructor is called, an exception is thrown?
  - The first one will be leaked

`make_shared` and `make_unique`

- make_shared<T> and make_unique<T> create an object and return an owning pointer
- The following two lines act the same
  - `auto ap = make_shared<T>(4, 7);`
  - `shared_ptr<T> ap = new T(4, 7);`
- make_unique wasn't added until C++14
  - Oops
- Now we can fix our previous example

```
void f()
{
   auto a1 = make_unique<A>(), a2 = make_unique<A>();
  // g(A *, A *) is responsible for deleting
   g(a1.release(), a2.release());
}
```

- *Effective Modern C++* Item 21
  - Prefer `std::make_unique` and `std::make_shared` to direct use of `new`

Let's improve it a little more

- If we can modify `g()`, we should really change it to take `unique_ptr<T>` arguments because otherwise, we would have an owning raw pointer
  - Remember, `g()` takes ownership, so it shouldn't use owning raw pointers
  - `g(unique_ptr<T>, unique_ptr<T>);`
- Now we can call
  `g(make_unique<T>(), make_unique<T>());`
- Interestingly, the following doesn't work because ownership will no longer be unique
  - ```
    auto p1 = make_unique<T>();
    auto p2 = make_unique<T>();
    g(p1, p2); // Illegal! unique_ptr not copyable
    ```
- To fix, we need to *move* from p1 and p2
  - `g(move(p1), move(p2)); // OK. unique_ptr is movable`
  - We'll learn about moving in detail next week
    ```

Getting raw pointers from smart pointers

- Sometimes when you have a smart pointer, you need an actual pointer
- For example, a function might need the address of an object but not participate in managing the object's lifetime
  - If you are not an owner of the object, there is no reason to use a smart pointer
- ```
  f(A *); // Doesn't decide when to delete its argument
  auto a = make_unique<A>{};
  f(a.get()); // a.get() gives you the raw A *
  ```
- Sometimes you want to extend the lifetime of an object beyond the lifetime of the `unique_ptr`.
- ```
  g(A *); // g will delete the argument when done
  auto a = make_unique<A>{};
  g(a.release()); // a no longer owns the object
  ```

What can go wrong with exceptions and what to do?

- The following slides, due to Andrei Alexandrescu, analyze different approaches to error handling
  - A name we will become very familiar with
- This will let us see a great example of how an expert analyzes a difficult problem
- Followed by a proposal (due to Botet, Bastien, and Wakely) to add a new approach called `std::expected`
- which was recently adopted for C++23

`std::expected` slides

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0323r12.html

HW 11-1

- Create a version of `make_unique` that doesn't throw exceptions

HW 11-2

- As mentioned in class, std::expected was not invented out of thin air
- Many similar approaches have been experimented with even at production scale
- Take a look at the AWS C++ SDK's `Outcome` class
  - https://sdk.amazonaws.com/cpp/api/LATEST/class_aws_1_1_utils_1_1_outcome.html
- How does it resemble C++23's `std::expected`?
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0323r12.html
- How does it differ?
- Where they differ, which do you like better and why?

HW 11-3: Extra credit

- Compare the version of `std::expected` in Andrei's slides with the one that was adopted into the language
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0323r12.html
- What changed?
- Why do you think it did?

HW 11-4: Extra Credit

- As mentioned in class, there was a lot of old-fashioned code in my solution to the hurricane problem
  - I've put it on Canvas
- Translate those into more modern C++
- Do you think it is an improvement?
- Why?