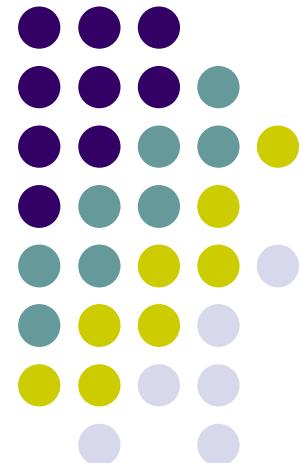


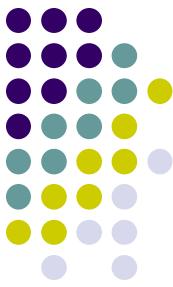
C++

January 5, 2023

Mike Spertus

mike@spertus.com





This week's lecture

- The first half of today's lecture is a survey of C++
 - Want to give some of the big picture without worrying too much about the technical details
 - Don't worry if some things are unclear or not covered in enough depth
 - We'll come back and start presenting language features systematically starting at "Hello, World" in the second half of the lecture



COURSE INFO



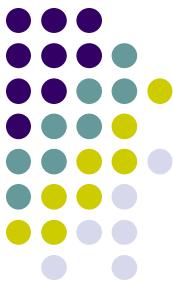
Who am I?

- Senior Principal in AWS Developer Experience team
 - Design tools for developing applications in the cloud: Both C++ and other languages
- Previously, Fellow at Symantec, where I was Chief Scientist for Cyber Security Services
- A long-time member of the ANSI/ISO C++ committee
 - Written over 50 C++ standards proposals, including C++11 [non-static data member initializers](#), and C++17' [Constructor Template Argument Deduction](#)
 - Consider joining the standards committee. It's the best way to learn C++ 😊
- Helped write one of the first C compilers for the original IBM PC AT
- Available at mike@spertus.com. Use it!



Useful Resources: Web

- The language standard.
 - Working draft at <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4892.pdf>
- <http://isocpp.org/>
- The C++ core guidelines is the best place to find out the currently recommended best practices
<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- <http://www.open-std.org/jtc1/sc22/wg21/>, the ISO committee website
- <http://herbsutter.com/> - Herb is the Convener of the standards committee
- <http://includecpp.org> - A great group support diversity in C++
 - You don't need to be in an excluded group to participate
- Online compilers, very useful
 - <https://wandbox.org>
 - Compiler Explorer at <https://godbolt.org>
- The C++ section of StackOverflow, cppreference.com, and r/cpp are all pretty good (cplusplus.com not so much)



Useful Resources: Books

- **Note:** No books required. Just if you find helpful
- Books from Bjarne Stroustrup, the inventor of C++. Be sure to use the latest editions
 - A Tour of C++ (3rd edition) – C++ for programmers, just like this course!
 - The C++ Programming Language (4th edition) – a thorough reference but only covers through C++14, so somewhat out of date
 - Programming: Principles and Practices – An intro to programming using C++. Don't be afraid to use it
- Nico Josuttis, *C++20, The Complete Guide*
 - As we will see, programming languages are as much about the standard library as the language itself
 - Covers C++20: Still very good but no filesystem, parallel algorithms, shared mutexes, user-defined literals for time, etc.
- Anthony Williams, *C++ Concurrency in Action* (2nd edition)
 - C++ multithreading through C++17. Our main topic for the last few weeks.
 - There is a free upgrade to 3rd edition at the Manning site
 - If you don't want to buy/read the whole book, Anthony's blog gives most of what you need in his multithreading in C++ series
 - <http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-1-starting-threads.htm>



Structure

- 3 hour lecture on Thursdays
 - Virtual for now
 - In person later (we hope)
- If you are ever stuck or have questions or comments
- Be sure to contact me
 - mike@spertus.com
- Or your TAs
 - Prashant Kumar: prashantk@uchicago.edu
 - Qingyang Xu: qingyangxu@uchicago.edu
- Office hours:
 - Mike: Thursdays, 3-5 JCL 398F and Zoom
 - Prashant: Monday, 4:30-6:30
 - Qingyang: Saturday, 1-3
 - Anyone: By arrangement



Homework and Lecture Notes

- Homework and lecture notes posted on Canvas Friday Morning
 - Choose MPCS 51044 and then go to “Pages”
- HW due on the following Thursday before class
- Submit on Gradescope
 - Expect an invite
- Graded homework will be returned by the start of the following class
- ☹
 - Since I go over the answers in class, **no late homework will be accepted**
- ☺
 - If you submit by Sunday midnight, you will receive a grade and comments back by Tuesday midnight, so you can try submitting again



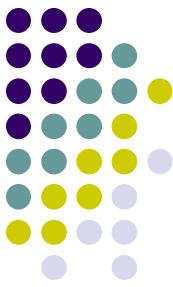
Discussion Group

- We use Ed
 - You can access from the Ed tab on Canvas
- If the question is directly about your homework solution, post it privately
- If it is more general, feel free to post publicly
- The interesting question is when to post



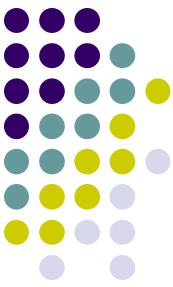
Programming in a Nutshell





Huh?

- As my T-Shirt today illustrates, programming is a process where you get stuck and then figure things out
 - Especially once you are working out of school
- Figuring things out when you're stuck is a skill we want to help you develop
- But we don't want you to just get frustrated spinning your wheels fruitlessly
- When you are ready to post
- Include what you have tried
- If you think you are close, it's encouraged to ask for a hint to get unstuck
- If you need more, don't be embarrassed to ask for that



Grading

- 2/3 HW
 - Many extra credit opportunities
 - Extra credit can get your HW total for the quarter to 100% (but no higher) to cancel out any problems you miss
- 1/3 Final
 - The biggest part of the final is to do a code review of a willfully bad (but unfortunately not worse than some code you'll see in real-life) program
 - Even though the final is 1/3 of the grade, it has at least as much impact on the final grade because the variance is higher



C++ for programmers?

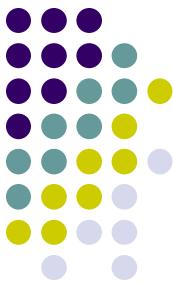
- Prior knowledge of C++ helpful but not required
- Expect you know how to program in some language
 - This is “not an introduction to programming” class
- Gloss over features that are similar to those in Java, C, etc.: `if`, `for`, `?:`, ...
 - OK if these are unfamiliar to you
 - Ask questions in class
 - Email or IM me or the Tas
 - Look up in the many recommended [texts](#)



C++ for programmers?

- This class concentrates on two things I consider equally important
- C++
 - As we will see, C++ is fundamentally different from other languages you might be familiar with, such as Java or C
- Advanced Programming
 - While most students have gotten a basic “how to program” intro from a class, friend, YouTube, etc., it’s hard to find material to systematically present advanced programming techniques that are so important to becoming an expert
 - For example, programming the cache is incredibly important in today’s multicore world and has a well-understood set of best practices, but very few programmers know them
 - This really doesn’t have much to do with C++ in particular, but what’s the point in becoming a C++ expert if you aren’t an expert programmer?
 - Indeed, the cache programming practices I will present are derived from a Java-based text
 - Risk reduction: The goal is that you will learn things from this class that you will use every day regardless of what language you use

What have you heard about C++?



- C++ is complicated and difficult to learn
- C++ is unsafe
 - You can scribble over memory and crash your program
- C++ is an incoherent mashup of every other programming language
- See [C++ considered harmful](#) for more



These are all true, but

- C++ often remains the language of choice
- In fact, it just moved ahead of Java to 3rd-place on the Tiobe Language Popularity index
- Big Data, Games, Fintech, ML, etc., C++ is even a distinguisher
- Why?

These are all true, but... ...that's a good thing



- At least for some use cases
- Most languages omit features that programmers often use incorrectly
 - Fewer bugs in code by unknowledgeable programmers
- C++ includes features if they support important use cases even if they can be misused by the uninformed
- The committee believes it is not unreasonable to expect professional programmers to spend time studying the tools they use every day
 - With C++, you won't bring a knife to a gunfight, but make sure you are expert enough to not shoot off your own foot
- Most of the advanced difficult features are for writing libraries. This can make writing applications in C++ easier than in other languages

Most devs learn C++ on the street and end up hating it



- It certainly didn't work for me
 - After over a decade of picking up C++ through general exposure and writing commercial C++ libraries, I thought I was an expert C++ programmer
 - Then I joined the C++ standards committee and found out I wasn't
 - The real inventors of C++ routinely used many powerful C++ techniques and idioms that I had never heard of
- Their libraries were much more powerful, flexible, performant, and easy to use than any I had produced are even imagined



C++ standards

- 1979 “C with classes” invented by Bjarne Stroustrup
- 1983 Renamed C++
- 1998 First standard.
- 2003 A minor standard revision Primarily fixed defects in the 1998 wording
- 2011 C++11 standard.
 - Called C++0X under development so “X” ended up being “b”! (We were overly ambitious)
 - Bjarne Stroustrup says it seems like a “Whole New Language”
- 2014 C++14 A minor release fixing sharp edges
- 2017 C++17
 - Medium-sized release mainly around library improvements: parallel algorithms, filesystem, string_view, optional, any, variant (tagged unions). Introduced Class Template Argument Deduction
- 2020 C++20
 - Major release centering around Concepts, a major rearchitecture of templates and std::format, which makes formatting no longer suck
 - Introduced important features like Coroutines, Ranges (a new approach to algorithms), and Modules (an importing mechanism), but not ready for prime-time use
- 2023 C++23
 - Ranges and Modules will become useful
 - std::expected is a new paradigm for error handling



Some huge gaps still remain

- No networking
- No thread pools
- Asynchronous programming broken
- No sender-receiver
- No reflection
 - Therefore no serialization, RPC, etc.
- Way too many things still require unsafe code
 - Let's try to understand this



Safety, or lack thereof

- C++ isn't safe
 - C++ lets you access any memory location directly by its address ("pointers")
 - Which is by definition unsafe
 - Many of the best known security CERTs are ascribed to unsafe memory usage in C and C++
- But it isn't as unsafe as people think either
 - Modern C++ language and best practices vastly improve the situation
 - We won't even mention pointers until nearly the end of the course
- But there is still a long way to go with many "unsafe by default" constructs
- We will dig deeply into this question throughout the quarter



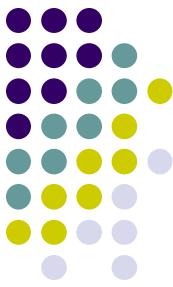
Which standard will we use?

- C++ programming has changed a lot over time
- C++20/23
 - Feels like a new language (Just like C++11 did)
 - One of the great things about C++ is that it stays on the forefront of language design, but can still run programs written in the 70s
- We want to learn C++ in a way that reflects the latest thinking about how to build applications
 - Also, speaking practically, when your org transitions to a modern C++ standard, you will be an instant leader if you already have a deep understanding of it
- However, compiler implementation is spotty and best practices for using some of the new features aren't yet well understood
 - Also, speaking practically, your org probably isn't using C++20 yet



Striking a Balance

- We will freely adopt better understood and widely available C++20 and C++23 features
 - For example, we will use C++20 Concepts from the start as a better way of writing generic code than traditional templates
 - But we will also provide workarounds or at least pointers for what to do in older C++ compilers
- For features that are still “teething” like modules, I will give expository lectures, but they won’t be a core part of what you will use



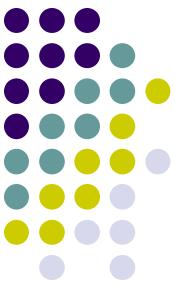
C++ is not a standard

- Just as important as knowing what the C++ standard says, you need to know what it doesn't say
- Even simple code relying on “obvious” non-standardized behavior may be very fragile
- Relying on non-standard C++ behavior is necessary. E.g.,
 - Bits in an integer
 - DLLs
 - Whether characters are signed or unsigned
 - “There are no interesting standards-compliant programs”
- However, it reduces portability and is fragile
- If you need to rely on non-standardized behavior (and you will), try to rely on “implementation-defined” rather than “undefined” behavior, so at least it is defined somewhere



What C++ isn't

- C++ isn't a better C or a worse Java
- Don't be misled by superficial similarities to C and Java.
- Good C or Java code is not necessarily good C++ code
 - >90% of C++ programmers make this mistake
 - One of my goals today is to convince you that this is correct
 - Knowing how to program in Java, C, or other languages does not mean you know how to program in C++



Characteristics of C++

- Compiled
- Multiparadigm
- Lightweight abstractions
 - Supports low-level system programming and high-level abstractions
- Statically typesafe and (largely) type inferenced
- Exceptions, Expected, and RAII



C++ is a compiled language

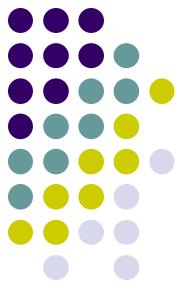
- Unlike languages like Java and JavaScript that use “Just in Time” compilation, C++ uses “Ahead of Time” compilation
 - Other popular compiled language are C, Rust, and Swift
- This can be less convenient because you need to manually compile your program before running it
- But the performance can be much faster as the compiler can take its time to analyze the program in depth
- It also enables “metaprogramming” the compiler to control its code generation, which will be a topic we will come back to



Multiparadigm

- C++ is not an object oriented language
- C++ supports object-oriented programming
- But it also supports other paradigms
 - Indeed, the standard library prefers compile-time dispatch (templates, see below) and only uses runtime dispatch in about a dozen places

C++ is a lightweight abstraction language



- That was the winner when Bjarne Stroustrup moderated a discussion on the C++ elevator pitch
- This is a mouthful, but I think the idea is right
- Programming languages typically fall into a tradeoff between being good for programmers (i.e., abstract) or good for computers (i.e., low-memory, fast, and low-level)
 - Abstract but slow/no low-level programming: Python, Java, JavaScript, ...
 - Efficient but not abstract: C, Assembler,
- The problem is, that we need both as performance vs abstraction constantly looms over almost all programming decisions
 - “Do I want to do it the clear modular, maintainable way or the performant way?”
- Unlike almost any other language, C++ gives you the best of both world, allowing you to create powerful abstractions that are lightweight by having all the abstraction “compiled-away” during the build using “templates”
- In other words, there is no performance penalty associated with using/creating abstractions

Delivering performance and abstraction



- C++ delivers performance by being a native language allowing low-level manipulation of code and data
 - No virtual machine, manual memory management, etc.
- C++ delivers abstraction through C++ generics, which use **compile-time computation** to generate optimal code
- If you want code that is
 - good for the programmer
 - clear, consistent, abstract, and extensible libraries
 - good for the computer
 - great time and space performance
 - low-level customizability
- Then C++ is most often the right language
- **Note:** Many programs don't need this. C++ has no equivalent of scripting or IPython notebooks that are great for small scripts where performance is not paramount



Objects

- We will need to talk about objects in the upcoming example
- We don't really know what an object is yet, but for now, just think of it as some structured data in memory
- An object can be a simple, contiguous region of memory, like a number

```
00 02 03 05 07 0b 0d 11 13 17 1b 1f 25 29 2b 2f 35  
10 3b 3d 43 47 49 4f 53 59 61 65 67 6b 6d 71 7f 83  
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Lightweight abstractions

Example: Copying objects

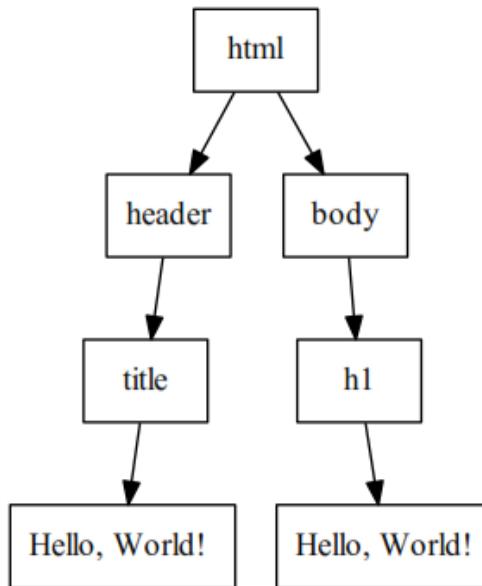


- How do we copy one data structure to another?
- In C, we get low-level performance by copying the underlying memory with the `memcpy` command
- While this is fast, it's not abstract, so it's not fit for production programs that are complex and need to be evolved over time
- Let's see why



“Compound” object

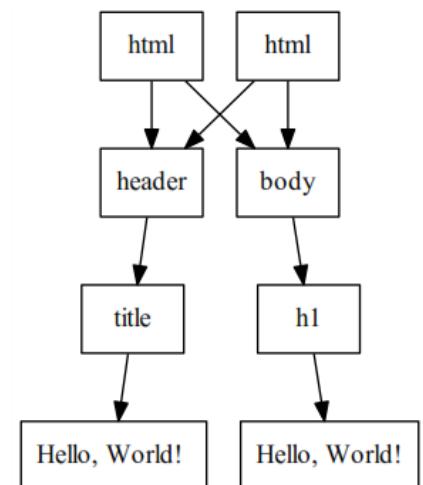
- More complex objects may (logically) contain other objects spread all over memory
- For example, an object representing an HTML page





Copying a compound object

- In C, if I copy a compound object, only the “root” is copied
- ```
HTMLPage a;
a = /* ... */; // "Hello" page
b = a; // Oops! Only copies root
```
- One generally has to manually write “deep copying” commands in C, polluting the code with brittle implementation details
- Can we do better in C++?





# Copying compound obj in C++

- We haven't talked about classes yet, but for now, they are just the way you create your own types in C++
- C++ classes let you specify a "copy constructor" that explains the right way to copy objects belonging to that class
- ```
class HTMLPage {  
public:  
    // Copy constructor does deep copy  
    HTMLPage(HTMLPage const &) { /* ... */}  
    /* ... */  
};  
  
HTMLPage a = /* ... */; // "Hello" page  
b = a; // Automatically does deep copy  
// Programmer doesn't care if storage is contiguous
```
- **Abstract:** Just copy any object with an assignment, independently of implementation
- **Lightweight:** The compiler generates the code, so it is just as efficient as-if you did it manually

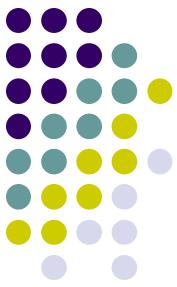


Lightweight abstractions

Example: Copying containers



- In Java, it is easy to create a new container that is a copy of an old container
- ```
// https://javarevisited.blogspot.com/2014/03/how-to-
clone-collection-in-java-deep-copy-vs-shallow.html
Collection<Employee> copy = new
HashSet<Employee>(org.size());
Iterator<Employee> iterator = org.iterator();
while(iterator.hasNext()) {
 copy.add(iterator.next().clone());
}
```
- This is very abstract, and doesn't care how the containers and objects are laid out in memory, so you can change from one type of container to another without breaking your code, etc.
- However, the price of this abstraction is poor performance as a contiguous array of primitive objects in memory is inefficiently copied one object at a time



# std::copy

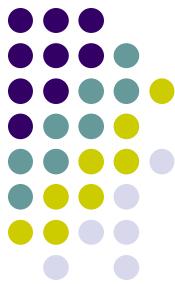
- C++ standard library provides a standard copy function that is the ultimate in abstraction
- It can copy from anything to anything else (anything from memory to containers to the console) and even does deep copies
- Example: Copying from a vector to array

```
vector<char> v;
```

```
...
```

```
copy(v.begin(), v.end(), arr);
```

# copy: A lightweight abstraction



- As abstract as Java
  - Can copy any collection to any other collection with no change to code
    - Less risk of bugs, separates implementation from usage, reduces brittleness so programs can evolve
- As efficient as C
  - The compiler uses a mechanism called *template programming* to automatically generate the most efficient code at compile-time, so if a block-move memory would do the trick, the compiler will simply generate the same `memcpy` as C
    - This results in an 800% performance improvement in such cases. Since copying objects is very common, this is not an unimportant optimization
- We will learn how to write code that does this over the course of the quarter

# Static type safety and inference



- If you are familiar with languages like Python and JavaScript, they are unsafe in a different way than we discussed earlier
- Any variable can hold any type of object
  - ```
var a = "foo"; // Typescript  
if(bar())  
    a = 7;          // a might change type  
console.log(a.length()); // Legal? Who knows?
```



C++ doesn't allow this

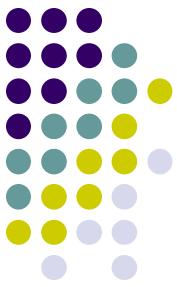
- The types are all checked during compilation
 - Static type-safety
- The compiler will not let you misinterpret the type of an object (unless you go to extremes to do so)
- `string a = "foo"s;`
`a = 7; // Ill-formed. Will not compile`



Static type safety is good

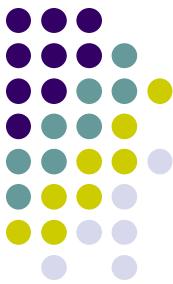
Programmers working in richly typed languages often remark that their programs tend to "just work" once they pass the typechecker

--- Benjamin Pierce, *Types and Programming Languages*



But at a price

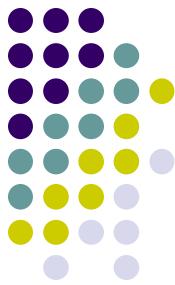
- Having to write down all of the types adds a lot of friction to programming
- That is why languages like Python and JavaScript feel so easy to just hack out some code
 - Even if it makes them less suited for reliable production systems



Solution: Inference

- C++ can often infer the type without you needing to mention it
- Giving you the benefits of static type safety without the headaches
- `auto a = "foo"s; // Deduces a is a string`
`a = 7; // Ill-formed. Will not compile`
- In fact, templates, which are one of C++' defining facilities, are based on inferring types
- We will look deeply at this

Exceptions, Expected, and RAI



- So much of programming is about error checking, handling, and cleaning up
- In C, this is estimated to take 40% of development time
- C++ has a rich array of lightweight abstractions to handle this (greatly enhanced in C++23)
- We will learn a lot about this in a few weeks, including what that crazy RAI acronym is



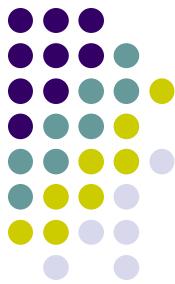
STARTING FROM “HELLO WORLD”

First things first. What compiler/IDE should you use?

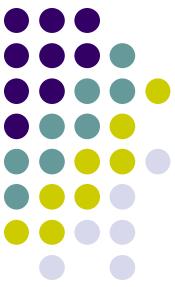


- We don't require you to use a specific compiler or IDE
- This has several advantages
 - You can get comfortable with the development environment and toolchain used by your org
 - We can learn as a class how portable C++ code is (Remember: C++ is not fully standardized)
- And one big disadvantage
 - You will have to install and configure it according to your choices (this can be harder than the actual programming!)
 - Don't hesitate to reach out for help

First things first. What compiler/IDE should you use?



- Any modern compiler with support for C++20 concepts
 - The latest versions of Clang, g++, and Visual C++ all fit the bill
 - Some of them don't default to the latest C++ standard, so you may need to set a flag like
`-std=c++20` (works on Clang/g++)
- Some popular IDEs that support these compilers are VSCode, XCode, Visual Studio, CLion, and Eclipse, as well as online IDEs like godbolt.org

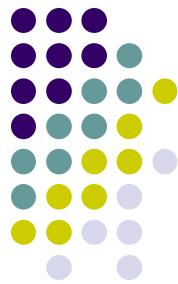


What is C++: Hello World

```
// https://godbolt.org/z/PvhWqG
#include <iostream>

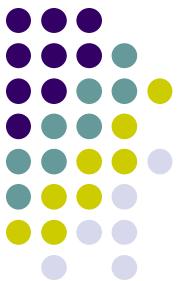
int
main()
{
    std::cout << "Hello, world!\n";
    return 0;
}
```

Let's do a more personalized greeting



```
#include<iostream>
#include<format>
using namespace std;

int
main()
{
    string name;
    cout << "What's your name? ";
    cin >> name;
    cout << format("Hello, {}!\n", name);
    return 0;
}
```



Does it compile for you?

- It does for me, but
- this is the C++20 question mentioned before
- We would really like to use the C++20 `<format>` library because the ancient formatted I/O mechanism is widely panned for good reason
- Unfortunately, not all C++ distributions implement `<format>`.



{fmt}

- Fortunately, there is a widely used open source library by the inventor that implements it
 - In fact, this was used to get the feature right
- While it's a little ugly to have to download a “standard” library, it's worth it in this case
 - as many C++ projects are already doing



Using {fmt}

- Download from <https://github.com/fmtlib/fmt> and add the include directory to your compiler's include path
- Modify the top of the file

```
#include <iostream>
#include <format>
#define FMT_HEADER_ONLY
#include <fmt/format.h>
using namespace std;
using namespace fmt;
```



The preprocessor

- `#include` and `#define` are commands to the C++ Preprocessor that does simple text cut and paste with no real knowledge of C++ at all
- `#include "foo.h"` cuts and pastes the contents of `foo.h` right into the including file
- `#define x 7` replaces all subsequent occurrences of the word `x` with the symbol `7`

How to think about the preprocessor



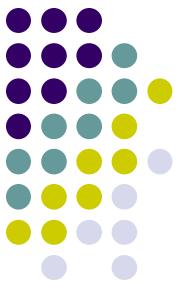
- While we're mentioning the preprocessor because we're need it now, don't be tempted to use it as a convenience for text manipulation
- Preprocessors cause lot of problems
 - Preprocessor definitions ignore namespaces
 - Having each including source file be ballooned to tens of thousands of lines by including libraries like `<iostream>` plays havoc with compile times
 - How should an IDE refactor?
- Newer languages like Java, Python, Scala, etc. do not include a preprocessor and C++ has been adding features (most notably modules) to eliminate the need to use the preprocessor



Sidebar: Why did we write `#define FMT_HEADER_ONLY`

- As a compiled language, C++ lets you separately compile libraries and link with them
- This can speed compilation, but make your build process more complex
- `{fmt}` can be used either as a linkable library or in “header-only mode,” where the preprocessor pastes in all the code that is needed, which we used for convenience (at the expense of longer compile times)

Of course, we could write this simple program without format



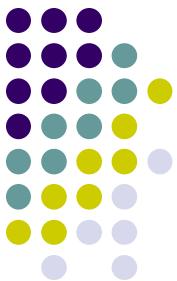
```
#include<iostream>
using namespace std;

int
main()
{
    string name;
    cout << "What's your name? ";
    cin >> name;
    cout << "Hello, " << name << "!\n";
    return 0;
}
```



Variables

- C++ has a variety of ways of defining and initializing variables, which we will cover in depth later
- For the moment, we simply mention the following
- ```
int i = 5; // i is an int initialized to 5
i = 7; // Now i is 7
i = "hello"; // Wrong type (C++ is not Python)
// Types can be inferred from initializer
auto j = 3; // j is an int
```

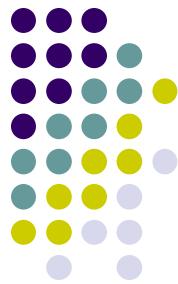


# Defining functions

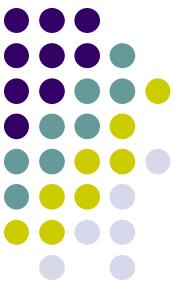
```
int square(int n)
{
 return n*n;
}

auto square(double n) // OK to have two functions with
{ // same name (overloading) as long
 return n*n; // as compiler can tell which you
} // mean by context
 // The type "auto" means the compiler
int main() { // should perform type inferencing
 cout << square(2) + square(3.1416);
 return 0;
}
```

# Statically typesafe largely type inferred



- The key to implementing lightweight abstractions is C++' powerful generic mechanism (also known as templates).
- Templates let you give a name to a not yet specified type
- Conceptually, it can often provide the simplicity of run-time types like in Python together with the safety of compile-time type validation
- This is one of the major themes in the evolution of C++ and is still a work in progress
- If you took my Big Data class last fall, this is similar in many ways to types in Scala

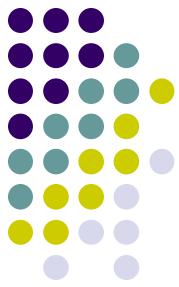


# Function templates

```
auto square(auto n)
{
 return n*n;
}
```

```
int main()
{
 return square(2) + square(3.1416);
}
```

# Template functions (pre-C++20)



```
// Should you be stuck with pre-C++20 tools
// at work, do this instead
template<typename T>
T square(T n)
{
 return n*n;
}

int main()
{
 return square(2) + square(3.1416);
}
```



# Containers: vector template

- The vector container in C++ is in fact a class template, so the compiler can build a version optimized for each class
  - `vector<int> v = { 1, 2, 3}; // type can be specified  
vector w = {1, 2, 3}; // or deduced also vector<int>  
// loops  
for(auto i : v)  
std::cout << i << ' '; // Prints 1 2 3`



# Let's look at some programs

- hello.cpp, hello\_personalized.cpp, vector\_simple\_demo.cpp, frame.cpp and frame\_2.cpp from Canvas
- The from programs are adapted and modernized from Koenig and Moo's *Accelerated C++* book

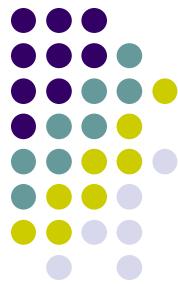


# HOMEWORK



# HW 1.1

- The purpose of this exercise is to help you make sure you have a suitable C++ compiler installed and that you know how to build programs and submit homework
- All referenced code is available on canvas
- Build, compile, and run the "Hello\_personal", "vector\_simple\_demo.cpp" and both "Frame" programs on your compiler of choice.
  - You may need to follow some of the steps from the lecture for enabling you to use format
- Send something to demonstrate that you've done this successfully (e.g., screenshots, any files you've written, including C++ files, makefiles, Visual Studio project files, a transcript of your shell session, etc.)
- Do not submit any executable files or large binaries! They aren't any good for the graders



# HW 1.2

- Print out the first 8 rows of Pascal's triangle.  
This assignment is most easily completed by using a nested container, e.g., a `vector<vector<int>>`.
- If you've never seen C++ vectors, look at `vector_simple_demo.cpp` on Canvas for a simple example(or ask me or a TA)



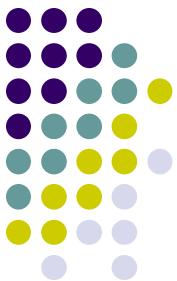
# HW 1.3 – Extra Credit

- For additional credit on the previous problem, it should be tastefully formatted. In particular
  - Use "brick-wall" formatting, in which the numbers of each row are presented interleaved with the numbers on the rows above and below.
  - The brick size should be the maximum size of any integer in the triangle. For aesthetic as well as technical reasons, it is useful if the brick size is odd, so you may increase the size by one if necessary to make this true.
  - Each number should be centered on its brick.
  - **Hint:** To get a number  $n$  centered in a field of width  $w$ , use `format(":{:^{} }", n, w);`



# HW 1.4 (Extra Credit)

- C++ tries to be compatible with C, but it's not perfect
- Let's see if you can break the compatibility
- Write a valid C program that is not a valid C++ program.
- Hint: There are ways to do this that don't require prior experience with C or C++. Look for some simple "thinking-outside-of-the-box" solutions.



# HW 1.5 (Extra Credit)

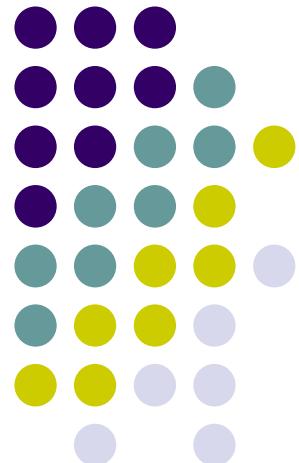
- Why is C++ called C++ and not ++C?
- Note: If you are new to languages with the “++” operator, see
  - <http://cplus.about.com/od/glossar1/g/preincdefn.htm>
  - <http://cplus.about.com/od/glossar1/g/preincdefn.htm>

# Advanced C++

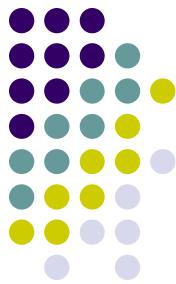
## January 12, 2023

---

Mike Spertus  
mike@spertus.com



# The relation between code and data

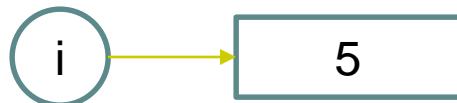


- A program refers to data objects through expressions in C++ (or some other language)
- When the program is run, the actual data objects exist in computer memory
- How does the code find, interpret, and manage the objects at runtime?



# Example: Simple variables

- Consider the following simple code
  - `int i = 5;`
- When the program creates the variable `i` at runtime, it allocates some memory (usually on the stack) to hold `i`'s data

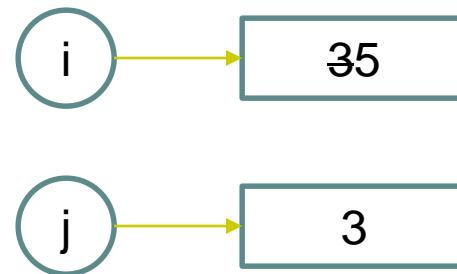


- Any expressions in the code involving `i` will use the data there



# Copying and assignment

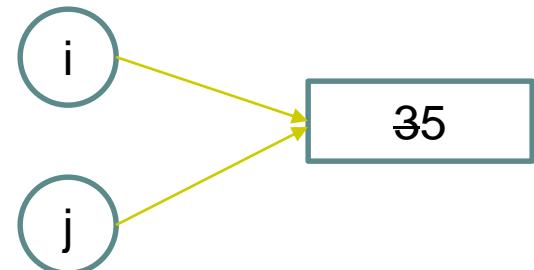
- In C++, when we do an assignment, the object is copied
- ```
int i = 3;
int j = i;
i = 5;
cout << j; // Prints 3
```
- https://godbolt.org/z/_GYLWx
- In this example, both `i` and `j` have their own storage associated with them in the running program
- **Warning:** In Java and Python, built-in types like `int` are copied, but user-defined types are not copied by assignment but are instead shared
 - cf next slide





References

- Suppose we wanted `j` to refer to the same data in memory as `i`, instead of a copy
- We can do that by creating `j` as a *reference* to an existing object (`int &`)
- ```
int i = 3;
int &j = i;
i = 5;
cout << j; // Prints 5
```
- <https://godbolt.org/z/z2GDSr>





# References vs values

- The only difference between references and values is that the value has its own copy of the data, and the reference reuses the object it is initialized with as its storage without making a copy
- In the example on the previous slide
  - i is a variable of type int and has its own storage
  - j is of type int& (“int reference”), and does not have any new storage associated with it. It just treats i’s storage as its own
- Both i and j can be used the same way after creation
  - If you are familiar with the use of & as the “address of” operator, this is an entirely different usage



# Warning: Object lifetime

- If multiple variables are referring to the same object, you have to be careful that the object still exists when you use it
- ```
int &f() // f is a function returning an int&
{ int i = 3; return i; }
```

```
int &j = f(); // f returns a reference to f's i,
              // so the int& j also uses that as
              // its storage
// However, the storage for f's i is released when
// f is done running, so j has bound itself to a
// no longer existing object!
cout << j;      // Undefined behavior!
```



Dynamic memory management

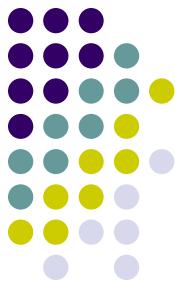
- As with C, C++ programmers have to manage the lifetime of objects explicitly
 - In contrast to garbage collected languages like Java
- If memory is released too early, the program could crash from trying to use an object that no longer exists in the computer's memory
- If memory is not released when it is no longer needed, the program can run out of memory (a memory leak)



Dynamic memory management

- While C++ has built-in commands to dynamically create and destroy objects
 - new and delete
- They should be rarely called
- Instead a “scoped ownership” model is used
- This is a special case of the most important idiom in C++, RAII, which we will learn much more about later

“Automatic manual memory management”



- In contrast to C, where memory management is very time consuming and error-prone, C++ has lightweight abstractions that, when used correctly, will automatically correctly managed the lifetimes of the object in memory
- The most common of these abstractions is `unique_ptr`



Creating a unique_ptr

- Calling `make_unique<T>` creates an object in memory on demand and returns a kind of handle to the object of type `unique_ptr<T>` that can be used to reference the object
- ```
// Create an int in memory and return
// an associated unique_ptr
unique_ptr<int> ui = make_unique<int>(5);
```



# A unique\_ptr

- Gives you access to the data in the object
  - When you need a reference to the object managed by the `unique_ptr`, use the `*` operator
  - `unique_ptr<int> ui = make_unique<int>(5);`  
`cout << *ui; // Prints 5`
- Manages the lifetime of the object
  - When the `unique_ptr` goes away, it will automatically free up the memory of the object that it is managing



# Updating a unique\_ptr

- If you bind a `unique_ptr` to point to a new object, it will free up the old one before it starts to manage the new one
- ```
auto ui = make_unique<int>(5);
cout << *ui; // Prints 5
ui = make_unique<int>(2);
cout << *ui; // prints 2
```
- The `unique_ptr` automatically releases the memory of the first object (the one with value 5) before it starts managing the new object (the one with value 2)



Transferring ownership

- In our example program later today, we will need to transfer ownership from one `unique_ptr` to another
- This is a little tricky
- Assignment doesn't work
 - `up2 = up1; // Oops! Two "unique" owners`
- Correct solution. Move from `up1`
 - `up2 = move(up1); // up2 is owner. Don't use up1`
 - For Rustaceans, this is like using assignment to transfer ownership in Rust
- We will learn more about moving soon
- But for now, you can treat it as a magic word

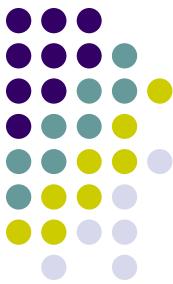
What about pointers, new, delete, malloc and free



- If you are a C programmer or pre-C++11 C++ programmer
- You might be familiar with constructs for working with memory at a low level like pointers, new, delete, malloc, and free
- We will cover these late in the course, but such low-level concepts should not be used outside of systems programming that has a specific need for low-level manipulation
- Normal programming should stick with lightweight abstractions like unique_ptr



CLASSES



Creating our own types

- If we could only use a language' built-in types and couldn't define any of our own, the power of the language would be very much restricted to what was built-in
- In my solution to the Pascal's triangle problem, I sort of created my own Row and Triangle types
 - using `Row = vector<int>;`
 - But what using does is let you give a new name to an existing type, so this is useful, but doesn't create genuinely new types



Classes

- To create your own type in C++, you must define a class
- Consider the following example (p. 61 in Koenig and Moo)

```
struct Student_info {  
    string name;  
    double midterm, final;  
    vector<double> homework;  
}; // Semicolon is required!
```

- See the sample “Grading” programs on Canvas
- Unlike C, no need for:
`typedef struct Student_info Student_info;`

Hey, that's a `struct`, not a class!



- That's OK, the only difference between a struct and a class in C++ is different default visibility of members.

- The following is equivalent

```
class Student_info {  
public:  
    string name;  
    double midterm, final;  
    vector<double> homework;  
};
```



Visibility of members

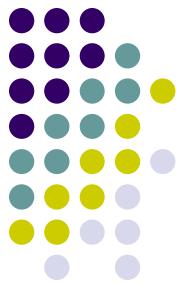
- public members are visible to everyone
- protected members are visible to subclasses
- private members are only visible within the class

Visibility (cont)



```
class A {  
    void f() {  
        cout << pub; // OK  
        cout << prot; // OK  
        cout << priv; // OK  
    }  
public:  
    int pub;  
protected:  
    int prot;  
private:  
    int priv;  
};  
class B : public A {  
    void g() {  
        cout << pub; // OK  
        cout << prot; // OK  
        cout << priv; // Error  
    }  
};  
void h(A a)  
{  
    cout << a.pub; // OK  
    cout << a.prot; // Error  
    cout << a.priv; // Error  
}
```

A class can have methods as well as fields

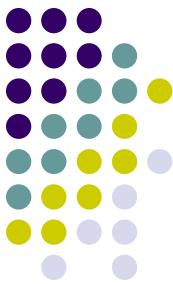


- You can also add member functions (methods) to go along with the data members (fields)

```
struct Student_info { // In header
    string name;
    double midterm, final;
    vector<double> homework;

    // Method to calculate the student's grade
    double grade() const {
        return (midterm + final + median(homework))/3;
    }
};
```

Another way to organize the code



- In the previous slide, the code for how to calculate the grade was put right inside the class definition
- It is also possible to put it in a separate file (to avoid cluttering the interface)

```
// In .h file
struct Student_info {
    string name;
    double midterm, final;
    vector<double> homework;
    double grade() const;
};
// In .cpp file
double Student_info::grade() const
{
    return (midterm + final + median(homework))/3;
}
```



Accessing methods and fields

- Use the . operator

- Student_info s;

```
s.name = "Mike";
```

```
s.midterm = 70;
```

```
s.final = 85;
```

```
s.homework.push_back(60);
```

```
s.homework.push_back(75);
```

```
cout << s.grade();
```



Static vs Dynamic types

- As we mentioned above, a program uses expressions to refer to objects in memory
- The static type is the type of the expression
 - Known at compile time
- The dynamic type is the type of the actual object referred to by the expression
 - May be knowable only at run-time
- Static and dynamic type generally only differ due to inheritance



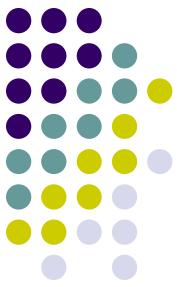
(Single) inheritance

- One can use inheritance to model an “isA” relationship
- `struct Animal { /* ... */ };`
- `class Gorilla : public Animal {...};`
- This means that a Gorilla “isA” Animal and can be referred to by Animal references



Static type vs Dynamic type

```
int i = 5; // S = int, D = int
Gorilla g; // S = Gorilla, D = Gorilla
Animal &a = g; // S = An&, D = Gor
Animal a2 = g; // Oops! Can't copy a Gorilla
                // into an Animal
unique_ptr<Animal> ua = make_unique<Gorilla>();
// Static type of *ua is Animal but Dynamic is Gorilla
ua = make_unique<Falcon>();
// Now S = Animal, D = Falcon
```



Virtual vs. non-virtual method

- A virtual method uses the dynamic type
- A non-virtual method uses the static type



Virtual vs. Non-virtual method

```
struct Animal {  
    void f() { cout << "animal f"; }  
    virtual void g() { cout << "animal g"; }  
};  
struct Gorilla : public Animal{  
    void f() { cout << "gorilla f"; }  
    void g() { cout << "gorilla g"; }  
    void h() { cout << "gorilla h"; }  
};  
void fun() {  
    unique_ptr<Gorilla> g = make_unique<Gorilla>;  
    Animal &a = *g;  
    a.f(); // Not virtual: Animal's f  
    a.g(); // Virtual: Gorilla's g  
    a.h(); // Error: h is not in animal  
    (*g).f(); (*g).g(); (*g).h(); // Gorilla's f, g, and h  
}
```



The `->` operator

- In the previous slide, we used clunky expressions like `(*g).f()` to call the `f` method of the object managed by the `unique_ptr` `g`
 - `(*g)` is a reference to the Gorilla object managed by `g`
 - `(*g).f()` calls its `f` method
- This is so common that there is a special shortcut notation for it
 - `g->f();` // Same as `(*g).f()`

Adding multiple grading strategies



```
struct Abstract_student_info { // In header
    string name;
    double midterm, final;
    vector<double> homework;

    istream& read(istream&);
    // Don't define grading strategy here
    virtual double grade() const = 0;
};
```

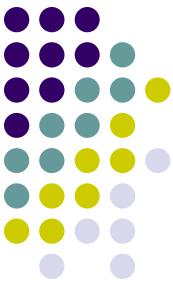


Here are a couple of strategies

```
struct BalancedGrading: public Abstract_student_info {  
    virtual double grade() const override {  
        return (midterm + final + median(homework))/3;  
    }  
};
```

```
struct IgnoreHomework: public Abstract_student_info {  
    double grade() const {  
        return (midterm + final)/2; // Ignore the HW  
    }  
};
```

How do we choose which grading strategy to use?



```
int main()
{
    unique_ptr<Abstract_student_info> si
        = make_unique<Balanced_grading>();
    si->read(cin);
    cout << "Grade is " << si->grade() << endl;
    return 0;
}
```



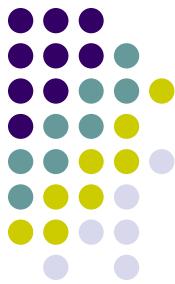
VIRTUAL METHOD PERFORMANCE

What are virtual functions and are they slow?



- Review: A virtual function is called based on the runtime type of the object even if it is accessed through a base class pointer
- You may have heard that virtual functions only have one more indirection, and you don't need to worry about the performance
- This is actually true most of the time
- But not always (this is not well-known)
- Let's look at an example

Virtual method implementation/performance



- Since the compiler doesn't know what the type is at compile-time, a virtual function is called through a reference to a table of functions that is stored in the object
- For a non-virtual method, the compiler knows what method will be called and calls it directly
- In general, this only adds a couple of clock cycles, so the cost of making a function virtual is usually negligible
- But that is not the whole story

Benchmark 1: Performance same if virtual



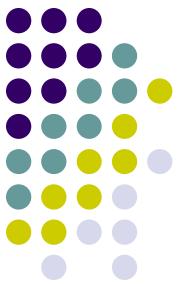
```
#include<chrono>
#include<iostream>
#include<format>
using namespace std;
using namespace std::chrono;

class A {
public:
    int f(double i1, int i2) {
        return static_cast<int>(i1 * log(i1)) * i2;
    }
};

int main()
{
    auto a = make_unique<A>();
    int ai = 0;
    auto start = high_resolution_clock::now();
    for (int i = 0; i < 100'000'000; i++) {
        ai += a->f(i, 10);
    }
    auto end = high_resolution_clock::now();
    cout << format("result of {} took {:.2f} ms\n", ai, duration_cast<milliseconds>(end - start));
    return 0;
}
```

Benchmark 2

Virtual is much slower



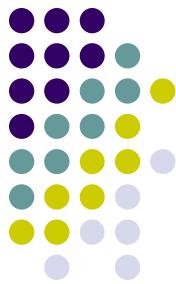
```
class A {  
public:  
    int f(double i1, int i2) {  
        return static_cast<int>(i1 * log(i1)) * i2;  
    }  
};  
  
int main()  
{  
    auto a = make_unique<A>();  
    int ai = 0;  
    auto start = high_resolution_clock::now();  
    for (int i = 0; i < 100'000'000; i++) {  
        ai += a->f(10, i);  
    }  
    auto end = high_resolution_clock::now();  
    cout << format("result of {} took {:.2f} ms\n", ai, duration_cast<milliseconds>(end - start));  
    return 0;  
}
```



What happened?

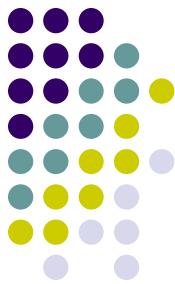
- Changing one line made virtuals 70 times slower than regular methods!
- The main performance cost of virtual functions is the loss of inlining and function level optimization
 - Not the overhead of the indirection
 - In the second benchmark, $\log(10)$ only needed to be calculated once in the non-virtual case.
- Usually not significant, but this case is good to understand
 - The more virtual functions you use, the less the compiler can understand your code to optimize it

Virtual methods: Best Practices



- Usually it is fine to make methods virtual, the performance cost is "almost always" minimal and the availability of runtime dispatch is powerful in object-oriented code
- But don't use virtual functions gratuitously if you are not using object-orientation, especially in performance critical or low level code
 - Virtual functions can impede compiler code optimization and unpredictably modify memory layout

Digression on benchmarking and optimization



- The first rule of optimization is

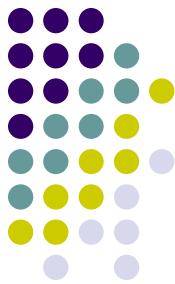
Don't!



Why not?

- There is no point in optimizing something that is not a bottleneck
- Highly-optimized code is generally harder to read, evolve, and maintain than regular code

How do I know what to optimize?



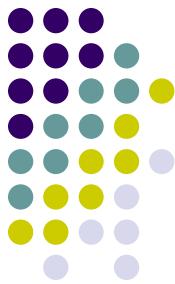
- Benchmark to find where your bottlenecks are
- And then decide whether they are worth optimizing



How to benchmark

- C++ has an excellent `<chrono>` library for working with time, that we will come to know and love
- We used its `system_clock::now()` method to get start and end times in our above benchmark
- Is this a good way to benchmark?
- Let's look at another example

Benchmarking the Fibonacci sequence



- This program calculates the Fibonacci series in the worst possible way (<https://godbolt.org/z/vY5d9bref>)

```
size_t
fib(size_t n)
{
    if (n == 0) return n;
    if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}

int const which{42};
int main()
{
    auto start = chrono::high_resolution_clock::now();
    auto result = fib(which);
    auto end = chrono::high_resolution_clock::now();
    cout << "fib(" << which << ")" is " << result << endl;
    cout << "Elapsed time is "
        << chrono::duration_cast<chrono::milliseconds>(end - start).count() << "ms" << endl;
    return 0;
}
```



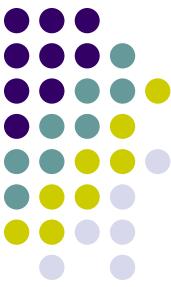
How fast is it?

- With clang
 - 2.34s
- With g++
 - 1.18s
- With MSVC
 - 0s



What!?

- The point is that the MSVC compiler reordered the code and got the end time before calculating the Fibonacci series



The "as-if" rule

- Compilers are allowed to behave differently from what the standard says
- As long as it behaves "as-if" it did things the right way
- It is OK to reorder as long as it doesn't bust a normative calculation
 - E.g., moving the use of a variable before its initialization
- Compilers (and hardware) do this extensively to improve performance
- Since the standard doesn't specify when things happens, moving a query of the current time is ok
 - Even though it busts our calculation 😞



The moral

- Be cautious about trusting timestamps
- Use a proper benchmarking tool
 - Google Benchmark
 - Nonius
 - gprof
 - ...



BACK TO CLASSES



Constructors

- `make_unique<Student_info>()` leaves `midterm`, `final` with nonsense values. (Use the original version. The one with the “pure virtual” method can’t be new’ed!)
- But not homework! We’ll understand that momentarily
- Fix as follows:

```
struct Student_info {  
    Student_info() : midterm(0), final(0) {}  
};
```



Order of construction

- Virtual base classes first
 - Even if not immediate
- First base class constructors are run in the order they are declared
- Next, member constructors are run in the order of declaration
 - This is why we didn't need to initialize homework. Vector's constructor creates an empty vector
- This is defined, but very complicated
 - Best practice: Don't rely on it
 - Good place for a reminder: Best practice: don't use virtual functions in constructors

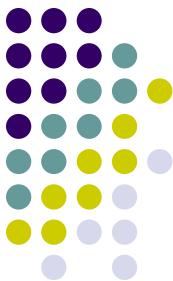


Constructor ordering

```
class A {  
public:  
    A(int i) : y(i++), x(i++) {}  
    int x, y;  
    int f() { return x*y*y; }  
};
```

- What is `A(2).f()`?

Answer: 18! (x is initialized first, because it was declared first. Order in constructor initializer list doesn't matter)



Avoiding spaghetti inheritance

- While the examples above allow us to use different grading strategies, it quickly gets out of hand
- Suppose we also had classes like `MPCS_Student_info` and `Undergraduate_Student_info` inherit from `Abstract_Student_info`
- But then to cover all combinations, we would also need `MPCS_IgnoreHW_Student_info`, `MPCS_BalancedGrading_Student_info`, etc.
- The number of classes grows exponentially, and soon we have “spaghetti inheritance”
- The class `NewStudent_info` in the example program shows a way to make more focused use of inheritance to avoid this



Sample program

- All of these classes are demonstrated in the Grading programs on canvas
- You will need to compile the .cpp files together in a single project
- For example

```
g++ -o Grading GradingTest.cpp Grading.cpp
```
- How to do this is compiler-specific, so you may need to check your documentation



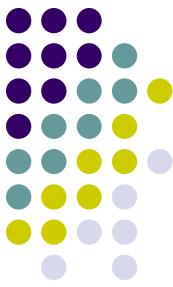
HW 2.1 (Part 1)

- Last week we mentioned `std::copy`. There is a related function in the `<algorithm>` header called `std::transform`, which lets you apply a function to each element before copying. Look up or Google for `std::transform` to understand the precise usage.
- Write a program that puts several doubles in a vector and then uses `std::transform` to produce a new vector with each element equal to the square of the corresponding element in the original vector and print the new vector (If you use `ostream_iterator` to print the new vector, you will likely get an extra comma at the end of the output. Don't worry if that happens).



HW 2.1 (Part 2)

- We will extend the program in part 1 to calculate and print the distance of a vector from the origin.
- There is also a function in the `<numeric>` header called `accumulate` that can be used to add up all the items in a collection.
 - (Googling for “`accumulate numeric example`” gives some good examples of using `accumulate`. We’re interested in the 3 argument version of `accumulate`).
- After squaring each element of the vector as in part 1, use `accumulate` to add up the squares and then take the square root. (That this is the distance from the origin is the famous Pythagorean theorem, which works in any number of dimensions).



HW 2.1 (Part 3)

- In real-life, you'd probably use `std::inner_product` to find the Euclidean length of a vector
- Learn about `inner_product` and use it to find a better way to accomplish part 2



HW 2.1 (Extra credit part 4)

- As yet another way to calculate the Euclidean length of a vector, is also a four argument version of accumulate that can combine transform and accumulate in a single step. Use this to provide another solution to part 2 of this problem



HW2.2

- One of the above slides referred to a function median, that takes the median of a vector of doubles.
- Part 1. Write the median function using the sort function in the algorithm header.
- Part 2. Write the median function using the partial_sort function in the algorithm header. Is this more efficient? Why do you think that? (You can give an intuitive or practical answer without precise mathematical analysis)
- Part 3. Write the median function using the nth_element function header. Do you think this is even more efficient? Why?
- Part 4 - Extra credit: Write a template function that can find the median of a vector of any appropriate type.
 - Although we haven't discussed writing our own template functions yet, looking at the template function for squaring from last week's slides
 - You can use any of the underlying algorithms from parts 1 to 3 above
- More extra credit. If there are an even number of elements, use the average of the middle 2 element



HW 2.3

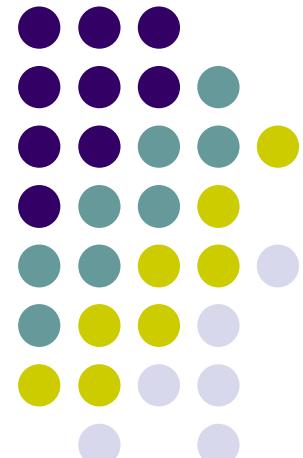
- Rewrite the pascal.cpp file in Canvas (or your own) to be class-oriented

C++

January 19, 2023

Mike Spertus

spertus@uchicago.edu



MASTERS PROGRAM IN
COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO



SORTING PERFORMANCE



Were you surprised by the different median performances?

- When I first built and run `median_test.cpp` (on Canvas), it showed that `nth_element` was the fastest, then `partial_sort`, and `sort` was slowest
- Seems logical
- “The less you need sorted, the less time it takes”
- Unfortunately, that is not correct

```
Microsoft Visual Studio... - □ ×
Using sort: 49.9692
38597ms
Using partial sort: 49.9692
27431ms
Using nth_element: 49.9692
2817ms
```



Oops! Forgot to optimize!

- Those last results were from a debug build, which is worthless for benchmarking



- If we switch to a release build, the results are much different
- Now `partial_sort` is the slowest!

```
Microsoft Visual St...
Using sort: 49.9692
1172ms
Using partial sort: 49.9692
3206ms
Using nth_element: 49.9692
165ms
```



What went wrong

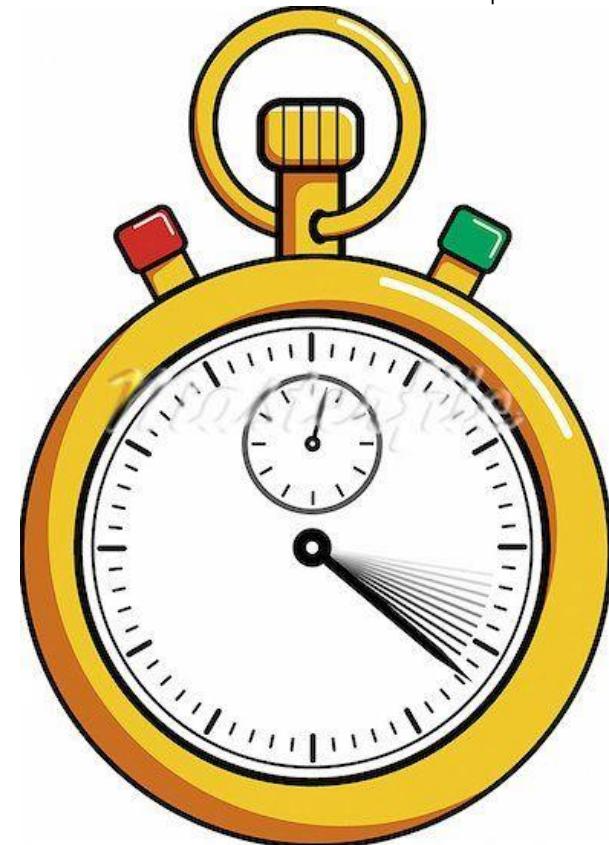
- Implementers use different algorithms for sort and partial_sort
- The problem with using partial_sort in the homework is that sorting half the range is pretty close to sorting all of the range, so the most efficient overall sorting algorithm is best
 - partial_sort is really designed for getting, say, the top 10 out of a million elements
- This isn't relevant to nth_element, and indeed its linear performance guarantee (28.7.2 [alg.nth.element]) gives the best performance

The moral: Always benchmark

Don't just assume



- Studies show that even expert software developers are bad at predicting the performance of code
- So measure, measure measure!





PASSING ARGUMENTS TO FUNCTIONS AND METHODS



Reference vs value

- We discussed reference and values for variables in Lecture 2, but it applies to function arguments as well
- In Grading.h, the method for reading HW scores has the signature
 - `istream& Student_info::read(istream&);`
- Likewise, in `class_oriented_pascal.cpp`, we have
 - `inline ostream& operator<<(ostream &os, Triangle triangle)`



Objects in code and data

- In program code, objects are used through expressions and variables in C++
 - E.g., $2^3 + 5$
- When the program is run, the actual data objects exist in computer memory
 - 11
- How are the C++ expressions and variables related to the actual objects in memory
 - Do they have the same type?
 - Do they have the same lifetime?
 - Where is the storage associated with an expression?

Associating storage and variables

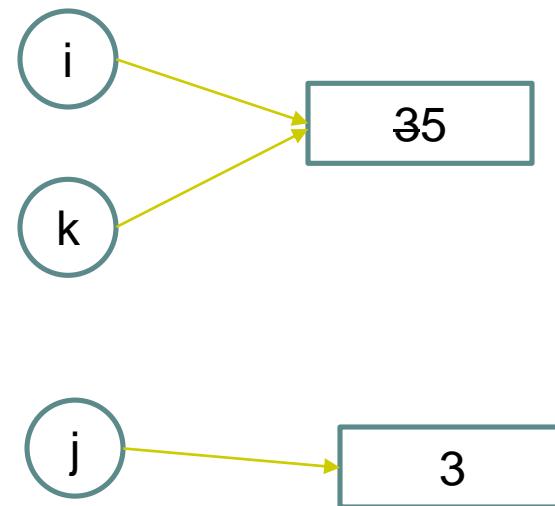


- If an expression or variable has a value type (i.e., not a reference type), it gets its own storage
 - Memory is allocated and a constructor is run
 - When the variable or expression leaves scope, its destructor is run and the memory for the object is released
- If the expression is a (lvalue) reference type, its storage needs to be supplied as part of initialization
 - No memory management, construction, or destruction occurs either during initialization or upon leaving scope
- Other than that, there is not much difference between values and references



Example

- ```
int i = 3;
int j = i;
int &k = i;
i = 5;
```

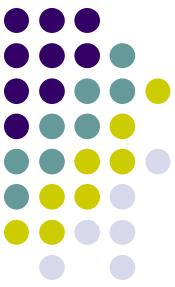


- See  
<https://godbolt.org/z/HsZN9> for an example with classes



# Passing arguments

- In C++, there are three ways to pass arguments to a function
- **By value**
  - `void f(X x);`
  - `x` is a copy of the caller's object
  - Any changes `f` makes to `x` are not seen by the caller
- **By reference**
  - `void g(X &x);`
  - `x` refers to the caller's existing object without creating a new copy
  - Any changes `g` makes to `x` also change what the caller sees
- **Spoiler: By move**
  - `void h(X &&x);`
  - We will learn about this soon



# Example:

```
void f(int x)
{ x = 3; cout << "f's x is " << x << endl; }
void g(int &x)
{ x = 4; cout << "g's x is " << x << endl; }

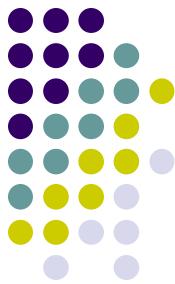
int main() {
 int x = 2;
 cout << x << endl; // prints 2
 f(x); // print's f's x is 3
 cout << x << endl; // prints 2
 g(x); // print's g's x is 4
 cout << x << endl; // prints 4
 return 0;
}
```



# Why does only C++ make you specify how an object is passed?

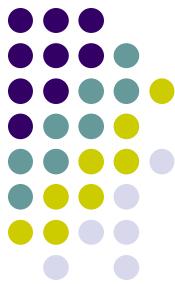
- Most languages only have only one way of passing arguments
- In C, arguments are always passed by value
  - The function always gets its own copy
- In Java, arguments are always passed by reference
  - The function sees the same object the caller passed
  - Exception: built-in numeric types like int are passed by value
- As usual, C++ let's you choose which you want (and adds a new “move” mode)

# Passing by value: pros and cons



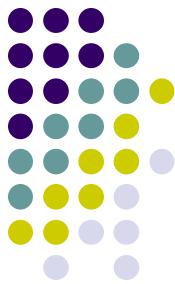
- **Pro:** It is very safe to call a function that takes its arguments by value because the function you called can't actually see the object you passed it
- **Con:** Copying an object may be very expensive
  - Imagine you are copying a map with a million key-value pairs
- **Con:** Doesn't work with inheritance/OO
  - If I copy an `Animal`, but it's really a `Gorilla`, my copy will only have `Animal` fields (and virtual methods may try to reference non-existent `Gorilla` fields, crashing my program!)
- **Con:** Sometimes I may want to modify the caller's object. E.g., they want me to clean the data in a vector.
- **Con:** Some objects are not copyable. E.g., `cout`, abstract types, `unique_ptr`

# Passing by reference: pros and cons



- **Con:** It is dangerous to call a function that takes its arguments by reference because it may unexpectedly modify your object
- **Pro:** You can fix this by passing the argument by “const reference”, which says the function is not allowed to modify it (e.g., by calling a non-const method on it)
  - void f(int const &i) { i = 3; // illegal: i is const }
  - You can say either `x const` or `const x`. Which is better? HW!
- **Pro:** Efficient, since object doesn’t need to be copied
- **Pro:** Works with inheritance/OO
  - No slicing because I am working with the original object
- **Pro:** I can modify the object if desired and appropriate
- **Con:** Managing memory is difficult
  - If my object has many owners, how do I know when it is safe to delete
  - We will learn best practices around this later

# So what should our Triangle printer look like?



- In the code, it is
- ```
ostream&
operator<<(ostream &os, Triangle triangle)
```
- What is best?
- We want to print to the original ostream, so better not copy it
 - In fact, the compiler won't even let us make a copy of an ostream
 - What would it even mean to have two cout's?
- Using ostream& is indeed best here
- Copying a (possibly big) Triangle just so our printing function can see it seems wasteful, so we would be better off with a reference
- But we also want to be able to print Triangles that we don't have the right to modify, which leads us to:
- ```
ostream&
operator<<(ostream &os, Triangle const &triangle)
```

# Should operator<< be a class member?



- It feels like it should, but the problem is that the first argument is `std::ostream&`, which means we would have to make it a member of class `std::ostream`
  - Oops, can't modify standard library classes
- So it is typically defined as a global
- However, there is an (advanced) way to put it inside the class
- C++ lets us put global functions inside the class definition as long as we precede it by `inline friend`
- See  
<https://stackoverflow.com/questions/10787655/c-friend-declaration-declares-a-non-template-function>



# Comparison with other languages you may know

|      | Free functions | Methods |
|------|----------------|---------|
| C    | X              |         |
| Java |                | X       |
| C++  | X              | X       |

| Code uses | Static type | Dynamic type       |
|-----------|-------------|--------------------|
| C         | X           |                    |
| Java      |             | X                  |
| C++<br>C  | X           | X (virtual method) |

| Parameter passing | By value | By reference | By move |
|-------------------|----------|--------------|---------|
| C                 | X        |              |         |
| Java              |          | X            |         |
| C++<br>C          | f(X)     | F(X&)        | F(X&&)  |

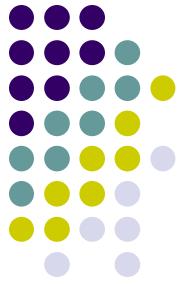


# SPECIAL MEMBERS



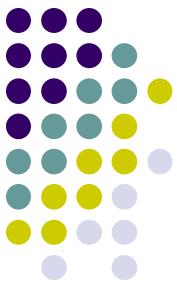
# What can go in a class?

- We have already talked about putting ordinary data members (*fields*) and member functions (*methods*)
- However, these are not the only items that a class can contain
- Which is what gives classes a lot of their power
- Let's see



# Static class members

- Usually, a class member depends on a class object
- ```
struct S { int i; double d };  
S s; // Need an s to  
s.i = 3; // access i
```



Static class members

- Sometimes you want the same class member to be shared by all objects of a class
- For example,

```
• struct CountedObject {  
    CountedObject() { objectCount++; }  
    ~CountedObject() { objectCount--; }  
    static size_t objectCount;  
};  
CountedObject c1, c2;  
cout << CountedObject::objectCount; // 2
```

Initializing static class members



- **Warning:** For technical reasons, you may have to initialize static data members with class types in a separate .cpp file
 - So its memory is only allocated once, not every time the header is included
- A.h
 - `struct A { static B ab; };`
- A.cpp
 - `B A::ab("xyz");`



Methods can be static too

- struct CountedObject {
 CountedObject() { objectCount++; }
 ~CountedObject() { objectCount--; }
 static size_t memUsed() {
 return objectCount * sizeof(CountedObject);
 }
 static size_t objectCount;
};

```
CountedObject c1, c2;
cout << format("{} objects using {} chars",
    CountedObject::objectCount,
    CountedObject::memUsed());
```

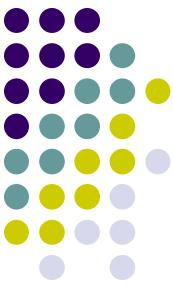


CONSTRUCTORS AND DESTRUCTORS

invoking a constructor to create an object

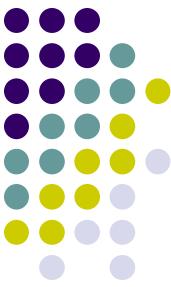


```
// construction.cpp on chalk
#include<initializer_list>
using std::initializer_list
struct A {
    A(int i, int j = 0); // #1
    explicit A(int i); // #2
};
struct B {
    B(int i, int j); // #3
    B(initializer_list<int> l); // #4
};
int main() {
    A a0 = 3; // A(3, 0) #1
    A a1(3); // A(3) #2
    A a1{3}; // A(3) #2. Uniform init: best practice but
    B b0(3, 5); // #3
    B b00 = {1, 2, 3, 4}; // #4
    B b1 = { 3, 5}; // #4
    B b2{3, 5}; // #4 Initializer list is preferred
}
```



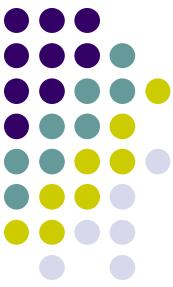
Constructor Signatures

```
struct A {  
    A(int _i = 0) : i(_i) {}  
    // Alternate  
    A(int _i = 0) : i{_i} {}  
    // Alternate 2  
    //    A(int _i = 0) { i = _i; }  
    int i;  
};
```



“Constructors” for fundamental types

- Since fundamental types don't have constructors, creating one without an initializer gives you garbage values
- `int i; // i could be anything`
- Reading uninitialized memory like this is a common source of bugs
- If you explicitly pass it zero constructor arguments, the memory is zero-initialized
- `int i{}; // i is Ø`



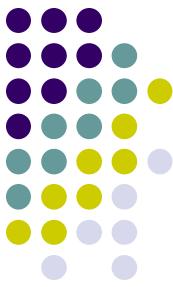
Special constructors

- To make it easier to create objects
- C++ will automatically generate some constructors for you
 - Default constructor
 - Copy constructor
 - Aggregate construction
 - Move constructor
 - Will cover in a later lecture



Default constructor

- Suppose I don't declare any constructors for my class
 - `struct S { int i; double d; };`
- Will it be impossible to construct an object of type S?
- No!
 - `S s; // Valid C`
 - That would break compatibility with C and feel punitive



Default constructor

- If you don't declare any constructors, C++ will create a no-argument constructor for you
- Even though S was declared as
 - `struct S { int i; double d; };`
- The compiler generates as if you wrote
 - `struct S {
 S() {}
 int i;
 double d;
};`

What if you don't want your own default constructor?



- For example, if I default-constructed an S object, I would get uninitialized memory
 - `S s; // s.i and s.d are garbage`
- One option is to write your own default constructor, and the compiler won't generate one
 - ```
struct S {
 S() : i(), d(1.2) {}
 int i; int j;
};
S s; // s.i == 0, s.j == 1.2
```

# Non-Static Data Member Initializers (NSDMI)



- While the previous slide works, it “feels” like the problem isn’t with S’ constructor but with i and d
- Fortunately, you can provide default initializers for members and the compiler-generated default constructor will do the right thing
  - ```
struct S {  
    int i = 0; double j = 1.2;  
};  
S s; // s.i == 0, s.j = 1.2
```
- For technical reasons, static data members cannot be initialized in class (hence the name)

What if you don't want a default constructor at all

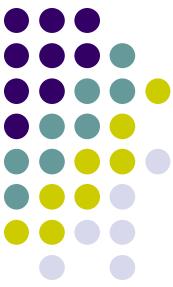


- Our Student_info constructor expected a name
- If the compiler generated a default constructor, you would get invalid nameless students
- Fortunately, if you define any constructor, the compiler won't generate a default constructor
 - `Student_info si1("Mike"); // OK`
`Student_info si2; // ill-formed! Will not compile`

What if you want a default constructor anyway?

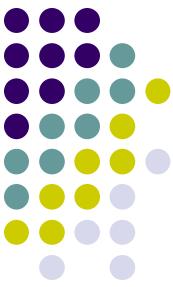


- Suppose we were OK with nameless students
- We could write a default constructor, but the compiler-generated one is really what we wanted, so we just say it has the default definition
- ```
struct Student_info {
 Student_info(string n) : name(n) {}
 Student_info() = default;
 name = "Mofei";
 /* ... */
};
```



# Copy constructors

- Besides default construction, you might construct a new S object by copying an existing one
  - `S s;`  
`S s2 = s;`
- Is this illegal?
  - It better not be, because it's legal C
- What constructor does it call?



# Copy constructors

- The compiler implicitly generates a copy constructor that copies all of the base classes and members (in the order described last week), as if you wrote
- ```
struct S {  
    S(S const &s) : i(s.i), d(s.d) {}  
    int i{}; double d = 1.2;  
};
```

What if the compiler generates the wrong copy constructor?



- While convenient, the compiler-generated copy constructor is not always correct
- For example, suppose you have a binary tree class
- The compiler-generated copy constructor would only copy the root, leaving us with tangled trees (see lecture 1 for a picture)
- Since the default implementation isn't right, write your own

- ```
struct BinaryTree {
 BinaryTree(BinaryTree const &b) { /* ... */ }
 /* ... */
};
```

# What if I don't want a copy constructor at all?

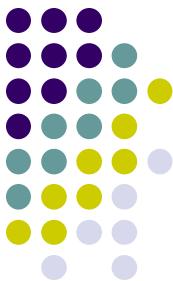


- One important design pattern is the “Singleton Pattern” that represents a class that only ever has one object
- For example, you may want to guarantee that all of your program uses the same Logger
- In this case, delete the copy constructor as below
- ```
struct Logger {  
    Logger(Logger const &) = delete;  
    static Logger myLogger;  
private:  
    Logger() { /* ... */; }  
    /* ... */  
};
```



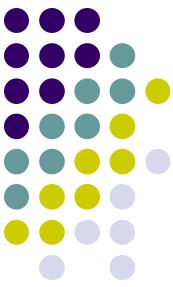
Aggregate construction

- What if the user wanted to create S with specific values for i and d?
 - `S s = { 5, 6.7 };`
- Is this legal?
 - It better be. It works in C
- What constructor is called?



Aggregate construction

- If a class is like a simple data structure with public members, C++ considers that to be an *aggregate* and generates a constructor that takes initializer
 - For the precise definition, see https://en.cppreference.com/w/cpp/language/aggregate_initialization
- In effect, S behaves as if it had the following constructor
 - ```
struct S { S(int const &i, double const &d) : i(i), d(d) {} ... };
S s = {1, 2.3}; // OK
S s2 = {.i =1, .d = 5.2 }; // OK. Designated initializer
```
- **TMI:** This is a little bit of an oversimplification as a constructor is not actually synthesized here for highly technical reasons, but “close enough for jazz.”



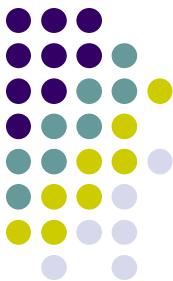
# Destructors

- Classes have “destructor” methods that do any needed cleanup when the object goes away
- When an object in memory goes away, its destructor is always called



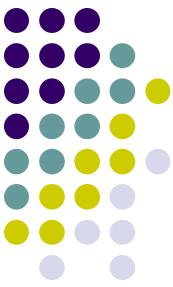
# The destructor for a class

- You can write a destructor for a class
  - ```
struct A {  
    ~A() {  
        /* Code to cleanup resources used by A */  
    }  
    /* ... */  
};
```
- If you don't write one, the compiler will generate it for you, called the default destructor
 - This will simply destroy all of the fields and base classes in the reverse of the order we gave for constructing them



Virtual destructors

- Suppose we have a `unique_ptr<Animal>` managing an object of dynamic type `Gorilla` at runtime
- Which destructor will be called?
- Same rule as other methods. If `Animal`'s destructor is not virtual, it will be called. If it virtual, then `Gorilla`'s
- **Best Practice:** Classes that are meant to be inherited from should have a virtual destructor. E.g.,
 - `virtual ~Animal() = default;`
- You may have gotten some warnings that our Grading program neglected this best practice
 - We didn't know about destructors yet!



Review: unique_ptr

- A `unique_ptr` is a class that manages an object in memory
 - It can give you a reference to the object
 - It cleans up the object it is managing when it goes away
- ```
void f() {
 // Create a unique_ptr with make_unique
 auto up = make_unique<vector<int>>({1, 5, 3});
 sort(up->begin(), up->end());

 // up cleans up the vector when you leave f
}
```

# How does unique\_ptr clean up memory?



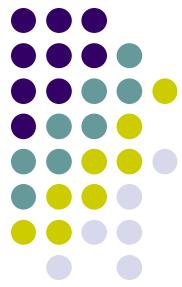
- Answer: unique\_ptr has a specially written destructor that calls the destructor of the managed object and then releases the object's memory
- This is why memory management takes 40% of C development time but little in C++
- This same technique is useful for many purposes, so it has a special name – RAII, which we will discuss later today

# We just discussed some special class members



- Static members
- Implicitly-generated default constructors
- Implicitly-generated copy constructors
- Implicit aggregate construction
- Implicitly-generated destructors

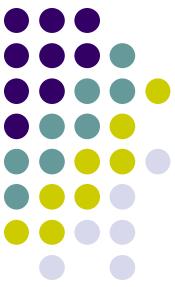
# There are some other special members



- Type conversions
- Overloaded operators
- Spoiler: Move constructors



# TYPE CONVERSIONS



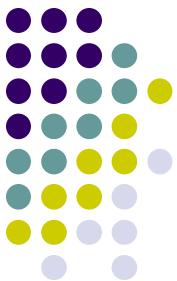
# Implicit conversions

- Built-in
  - `int i = 780;`  
`long l = i;`  
`char c = 7;`  
`char c = i; // No warning, but dangerous!`
- Polymorphism
  - `unique_ptr<Animal> ap = make_unique<Dog>("Champ");`
  - `void f(Dog &dr) { Animal &ar = dr; }`
- User-defined
  - Constructors
  - Operator overloading
- “Standard Conversions”
  - Defined in clause 4 of the standard



# Constructors and typecasts

```
struct A {
 A();
 A(int i); // Creates a type conversion
 A(int i, string s);
 explicit A(double d); // Does not create a conversion
};
A a;
A a0(1, "foo");
A aa = { 1, "foo" };
A a1(7); // Calls A(int)
a1 = 77; // ok
A a3(5.4); // Calls A(double)
a3 = 5.5; // Calls A(int)!!
```



# Type conversion operators

```
struct seven {
 operator int() { return 7; }
};

struct A { A(int); }

int i = seven();

A a = 7;

A a = seven(); // Ill-formed, two user-defined
// conversions will not be implicitly composed
```



# Explicit conversions

- Old-style C casts (Legal but bad!)
  - `char *cp f(void *vp) { return (char *)vp; }`
- New template casting operators
  - `static_cast<T>`
    - Like C casts, but only makes conversions that are always valid. E.g, convert one integral type to another (truncation may still occur).
  - `dynamic_cast<T&>`
    - Casts between reference types. Can even cast a `Base&` to a `Derived&` but only does the cast if the target object really is a `Derived&`.
    - Only works when the base class has a vtable (because the compiler adds a secret virtual function that keeps track of the real run-time type of the object).
    - If the object is not really a `T &`, `dynamic_cast<T&>` throws `std::bad_cast`;
  - `reinterpret_cast<T*>`
    - Does a bitwise reinterpretation between any two pointer types, even for unrelated types. Never changes the raw address stored in the pointer. Also can convert between integral and pointer types.
  - `const_cast<T>`
    - Can change constness or volatility only. There are usually better alternatives



# OPERATOR OVERLOADING



# Operator overloading

- You can overload operators just like functions
- The following operators can be overloaded:
- Unary operators:

+ - \* & ~ ! ++ -- -> ->\*

- Binary operators:

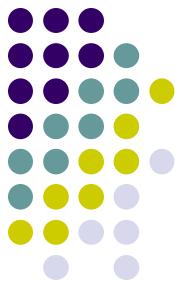
+ - \* / % ^ & | << >>  
+= -= \*= /= %= ^= &= |= <<= >>=  
< <= > >= == != && ||  
, [] () <=>  
new new[] delete delete[]

# Which operators can't be overloaded?



- ., .\* , ?: , ::
- Fame and fortune await for the one who figures out how to overload “operator.()”

# Sometimes you have a choice whether to use a special member



```
class myString {
 myString(const char *cp);
 char operator[](size_t idx) const;
 myString operator+(myString const &addend) const;
 myString operator+=(myString const &addend);
};

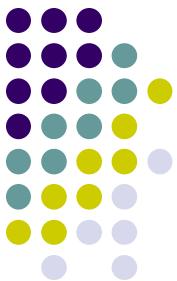
// Alternatively
myString
operator+(const myString &s1, const myString &s2);

myString
operator+=(const myString &s1, const myString &s2);
```

# Which way of overloading addition is better?



- Consider "Hello" + myString("World")
- Doesn't work for the member function
  - The first argument isn't even a class, so the compiler wouldn't know where to look for a member function.
- What about myString("World") + "Hello"
  - Works for both
- Using a global function makes sure both arguments are treated the same way, which fits the intuition that addition operators, which are generally commutative, should apply the same rules to each arguments.



# Do the same way for printing

- ostream &

```
operator<<(ostream &os,
 myString const &ms) {...}
```

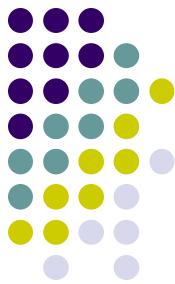
```
myString ms("foo");
cout << ms;
```

# Revisiting our Pascal's triangle



- In our Pascal's triangle implementation, we wrote an `ostream` insertion operator
- `ostream& operator<<(ostream &, Triangle const &);`
- In our original implementation, `Triangle` was a type alias (synonym) for `vector<vector<int>>`
- This worked for Triangles, but it also means that someone trying to print a `vector<vector<int>>` that is not a Pascal's triangle will be surprised that it is printed as a Pascal's triangle
- By making `Triangle` as class, it ensures `operator<<` will only be used for printing Pascal triangles

# Advanced: Printing to general output streams

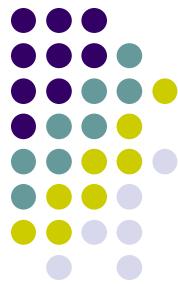


- As we will see later, ostream is really an instance of a class template  
`using ostream = basic_ostream<char, char_traits<char>>`
- To be able to print to any output stream, use a function template
- `auto & operator<<(basic_ostream<auto, auto> &os,  
 A const &ms) { os << 5; return os;}`

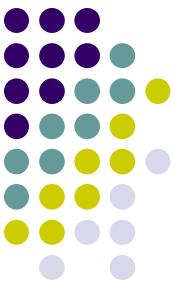
```
void f()
{
 A a;
 cout << a;
 wcout << a; // OK
}
```

- <https://godbolt.org/z/8q7f5b>

# How does an I/O manipulator get invoked

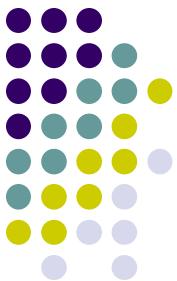


- Recall that endl is defined (as modulo some template complication that is irrelevant here) follows
- ostream &  
endl(ostream &os)  
{  
    os << '\n';  
    os.flush();  
    return os;  
}
- How come “cout << endl;” actually behaves as “endl(cout)”?



# Another overload!

```
ostream &
operator<<
(ostream&os,
 ostream&(&manip)(ostream &))
{
 return manip(os);
}
```



# How does a unique\_ptr work?

- Overloading operator->() and operator\*() of course
- operator->() overloads with a unique rule
  - Keep doing -> until it is illegal
- As mentioned above, the destructor calls the destructor of the managed object and releases its memory



# THE C++ OBJECT LIFECYCLE



# Object duration

- Automatic storage duration
  - Local variables
  - Lifetime is the same as the lifetime of the function/method
- Static storage duration
  - Global and static variables
  - Lifetime is the lifetime of the program
- Dynamic storage duration
  - Lifetime is explicit
  - Created with commands like “make\_unique”
  - Programmer controls when it ends
- In all cases, the constructor is called when the object is created and the destructor is called when the object is destroyed

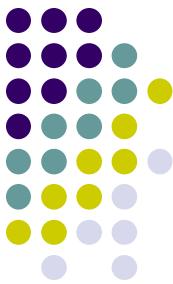


# Automatic duration objects

- Automatic objects are destroyed at the end of their scope

```
struct A {
 A() { cout << "A() "; }
 ~A() { cout << "~A()"; }
};
struct B {
 B() { cout << "B()"; }
 ~B() { cout << "~B()"; }
};
void f() {
 A a;
 B b;
}
int main() { f(); return 0; }
// Prints A() B() ~B() ~A()
```

# Members have automatic duration too



- ```
struct A {
    A() { cout << " A() "; }
    ~A() { cout << " ~A()"; }
};

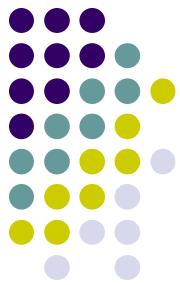
struct C {
    C() { cout << " C()"; }
    ~C() { cout << " ~C()"; }
    A a; // Goes away when C does
};
void f()
{
    C c;
} // Prints A() C() ~C() ~A()
```



Static duration objects

- These objects have the same lifetime as the program
- They are created (and their constructors called) when the program starts
 - In the order they are declared within a file
 - Order is undefined if they are in separately linked files
 - Even before `main()`
- They are destroyed (and their destructors called) when the program ends
 - Right after `main()` returns

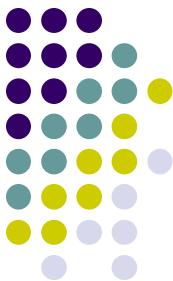
Creating static storage duration



```
int i; // Created at program start
struct A {
    static int j; // Created at program start
    void f() {
        static int k{}; // Created first time f() is called.
                        // Does not lose its value between calls
    }
};

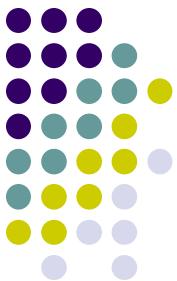
A a; // Created at program start

static A a2; // Created at program start
// a2 not visible outside of current translation unit
void g() {
    static A a3; // Created first time g() is called
}
```



Function-static lifetimes

- A static variable in a function is initialized the first time the function runs
 - Even if the function is called from multiple threads, the language is responsible for making sure it gets initialized exactly once.
 - If the function is never called, the object is never initialized
 - As usual, static duration objects are destroyed in the reverse order in which they are created



Dynamic duration objects

- These are created and destroyed under the control of the program
- Dynamic objects allow you to write programs whose data structures can be determined at runtime based on user input, etc.
- As we've mentioned, you can control their lifetimes with a `unique_ptr`



ILLUSTRATIVE LIFETIME EXAMPLE



Static storage duration

```
#include <iostream>
using namespace std;
struct A {
    A() { cout << "Creating an A object" << endl;}
};
A static_a;
```

```
int main()
{ ... }
```

- Prints "Creating an A object" before main is run because all global and static objects need to be constructed before starting the main program.

Case study: Why is cout safe to use?



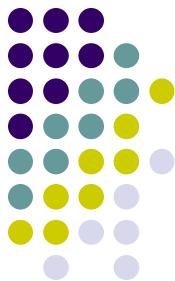
- There's something a little worrisome here. cout is a global object defined in the C++ run-time library. The <iostream> header declares it as:
`extern ostream cout;`
- How do we know cout will be initialized before static_a?
- Remember, order of static initialization is undefined for global objects defined in different source files
- Advanced sharp corner but good for getting practice with object lifecycles

When does the global variable cout get constructed?



- If the standard library ignored the issue, it might or might not work, depending on whether cout or static_a is initialized first.
 - Unacceptable for static constructors not to be allowed to write to cout.
- Fortunately, there is a static method ios_base::init() that initializes the standard streams.

Can we force cout to be initialized before static_a?



- Sure, use a static constructor ourselves

```
#include <iostream>
using namespace std;
struct ForceInitialization {
    ForceInitialization() { ios_base::Init(); }
};
ForceInitialization forceInitialization;
struct A {
    A() { cout << "Creating an A object" << endl; }
};
A static_a;
...
```



Abstracting into a header

- We will need to include ForceInit in any file that might use cout during static initialization, so extract it into a header ForceInit.h

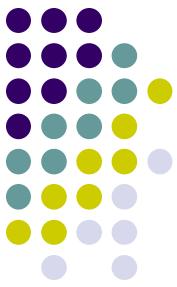
```
#include <iostream>
struct ForceInit {
    ForceInit() { std::ios_base::Init(); }
};
```



Abstracting into a header

- Oops, if the header is included multiple times in our .cpp file (including indirectly from other files we include), we will inadvertently call Init() twice!
- We can fix with an (ugly) include guard
 - Modules will fix right as we will learn later
 - Most compilers also support #pragma once
-

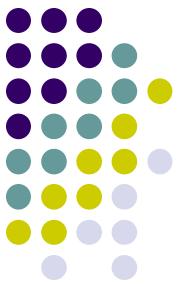
```
#ifndef FORCE_INIT_H
#define FORCE_INIT_H
#include <iostream>
struct ForceInit {
    ForceInit() { std::ios_base::Init(); }
};
static ForceInit forceInit;
#endif
```



Static vs. Global

- In the file above, we needed to make `forceInit` static, so multiple files didn't define the same global variable.
- However, the previous file still isn't right because `ios_base::Init()` will be called once for each source file, and we only want to call it once.

Preventing multiple initialization



```
#include <iostream>
namespace csp51044 {
    struct ForceInit {
        ForceInit() {
            if(count == 0) {
                count = 1;
                ios_base::Init();
            }
        }
    private:
        static int count;
    };
    static ForceInit forceInit;
```



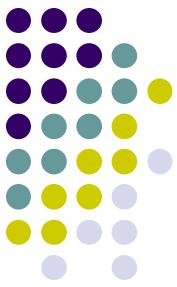
So useful, it's already there

- This idiom is extremely useful, and is actually part of the iostream header, so as long as you include iostream above where you use cout, you're (almost) OK

What if we want our object to outlive the automatic scope?

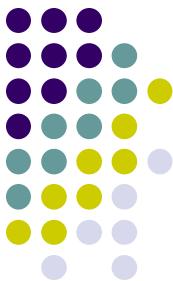


- It might be that we want our dynamic object to be longer-lived than the `unique_ptr` that is managing it
- Just transfer ownership to another `unique_ptr`
 - `up2 = move(up);`
- This allows us to chain into new scopes while always maintaining an owner of the object
- What is that mysterious move function and what are those move constructors I mentioned?
- Stay tuned next week



HW 3.1

- In slide 30, we mentioned that overloads of operator`+=` can be either class member functions or global functions
- Which one do you think is better and why?



HW3.2

- Use static duration objects to write a program that prints “Hello, world!” with the following main function:

```
int  
main()  
{  
    return 0;  
}
```

- Extra credit—Give a solution that depends on constructor ordering. The more intricate the dependence, the greater the extra credit.



HW 3.3: Extra Credit

- An object of a class that implements operator() is called a *functor*. Functors are objects that can be used as if they were functions.
- For a simple (but useless) example of the syntax, you can look at

https://en.cppreference.com/w/cpp/language/operators#Function_call_operator

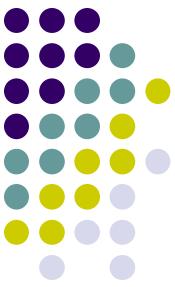
- For your homework, define a Nth_Power functor type so that code like the following prints cubes.

```
int main()
{
    vector<int> v = { 1, 2, 3, 4, 5 };
    Nth_Power cube{3};
    cout << cube(7) << endl; // prints 343
    // print first five cubes
    transform(v.begin(), v.end(),
              ostream_iterator<int>(cout, ", "), cube);
}
```



HW 3.4 – Practice with classes

- Your assignment is to implement the "Animal Game." The idea is that you chose a secret animal. The computer then asks you questions about the animal, terminating with a guess. If the guess is right, the computer wins, if it is wrong you win. But as part of winning, you have to provide your animal, and a differentiating yes/no question that the program can use to learn more animals
- See, <http://www.animalgame.com/> for an example. Your version will start knowing just one question and two animals and will be text only
- We have used `>>` to get input from `cin`, but that only reads one word, so you may want to consider using `readline`.
- Extra credit: In order to get more practice with object lifetimes, add a “forget” command to reset the program to its initial state



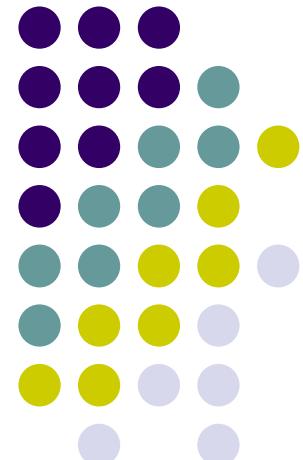
HW 3.5 – Extra Credit

- `const X` and `X const` mean the same thing
- Which do you think is better and why?

C++

January 26, 2023

Mike Spertus
mike@spertus.com



MASTERS PROGRAM IN
COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO



Agenda: C++ Algorithms

- We have explored standard library algorithms like transform, accumulate, and inner_product
- There are many more, some of which we'll discuss below

Agenda: General algorithm topics



- C++ algorithms are designed to interact with lambdas
 - We will take this opportunity to cover lambdas in more depth
- There are some problems with the model of C++ algorithms (mainly lack of compositionality)
- We will learn about how this can be addressed going forward



CALLABLES



Callables

- In our algorithms, it is often good to pass things that can be called
 - We saw this as far back as our week 2 homework on inner products
- But what can be called?
- Functions of course
- But there are also a bunch of other things that are useful to call
- Let's look in more detail



Functions

- double f(int i) { return 3.2 * i; }



Lambdas

- As we saw earlier in this course, besides named functions, you can also call lambdas
- Here is one that squares its integer argument
- `[] (int x) ->int { return x*x; }`



Lambda return values

- If your lambda does not contain return statements that return different types, the compiler can usually figure out the return type without your saying anything

```
[] (int x, int y) {  
    int z = x + y; return z + x;  
}
```

- If needed, you can explicitly specify the return type with the following notation

```
[] (int x, int y) -> int {  
    int z = x + y; return z + x;  
}
```



Capture lists

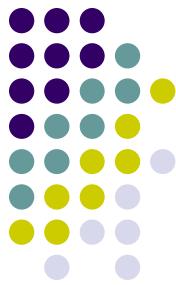
- To capture local variables by reference, use [&]
- To capture local variables by value, use [=]
 - ```
auto f() {
 int i{3};
 auto lambda = [=](){ cout << i; }; // lambda has a copy of i
 return lambda;
}
// Even if the original i has gone away, you can still run lambda
f()(); // prints 3 even though original i no longer exists
```
- C++ annoyingly makes `const` copies, so they may be hard to modify.  
You can turn this off with the keyword `mutable` as follows
  - ```
int i{3};
auto lambda = [=]() mutable { cout << i++; }; // lambda has a copy of i
// Note: Only modifies lambda's i, not the original variable
```
- You can write fancier capture rules if necessary
 - See <https://en.cppreference.com/w/cpp/language/lambda> for details



Lambdas and algorithms

- Now all of the standard library algorithms can be used as easy as for loops
- ```
std::vector<int> someVec;
int total = 0;
std::for_each
(someVec.begin(), someVec.end(),
[&](int x) { total += x; });
```

# Lambdas are not necessarily functions



- While lambdas act a lot like functions, they may not actually be functions
- ```
int i = 7;
auto x = [&i](int j) { return i*j; }
/* ... */
cout << x(5);
```
- x isn't really a function because it doesn't just depend on its argument j , it also depends on the captured reference to i

How are lambdas implemented then?



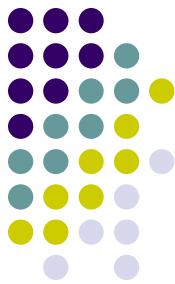
- Lambdas are actually implemented as classes that overload operator()
- The lambda in the last slide actually has a type that looks something like
- ```
struct X {
 X(int &i) : i(i) {} // Captures i
 int operator()(int j) {
 return i*j;
 }
 int &i;
};
```



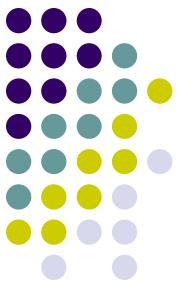
# Polymorphic lambdas

- Lambdas can take auto parameters and inference them
- `[](auto x) { return x*x; }(7); // 49`
- `[](auto x) { return x*x; }(7.5); // 56.25`
- How does this work?
- Again, the type of the above lambda is a class with an overloaded operator()
- ```
struct polymorphicSquare {
    auto operator()(auto x) { return x*x; }
};
```

Using a lambda with an algorithm



- Let's say we wanted to print the number of even members of a vector
 - With a for loop
 - ```
vector v = {2, 5, 6, 9};
size_t evenCount = 0;
for(int i: v)
 if(i % 2 == 0) evenCount++;
cout << evenCount;
```
  - With an algorithm
    - ```
vector v = {2, 5, 6, 9};  
cout << v.count_if([](int i) { return i%2 == 0; });
```



Captures can be helpful too

- for loop to draw a hand of cards for each player
 - CardDeck d;
vector<Person> players;
for(auto p: players) {
 p.hand = d.draw();
}
- With algorithm
 - CardDeck d;
vector<Person> players;
for_each(players, [&](auto p) {p.hand = d.draw(); })



Functors

- If you write your own class with an overloaded operator(), it's called a functor
- This allows you to create stateful objects that can be called like functions
- ```
struct Accumulator {
 int() (int j) { i = i + j; return i }
 int i = 0;
};
Accumulator acc;
cout << acc(1); // Prints 1
cout << acc(2); // Prints 3 = 1+2
cout << acc(4); // Prints 7 = 1+2+4
```
- As shown in the previous slides, lambdas are often implemented as functors

# Summary: A lot more things than functions can be called



- We have just discussed function pointers, but in C++, functions aren't the only thing that can be called
  - Call a function
  - Call a lambda
  - Call a functor



# More C++ algorithms

- `all_of`
- `any_of`
- `none_of`
- Check if all/any/none of the items in a container (or range) have a certain property



# More algorithms

- `copy_n`

```
vector<int> v = getData();
// Print 5 elements
copy_n
 (v.begin(), 5,
 ostream_iterator<int>
 (cout, "\n"));
```



# More algorithms

- `find_if_not`

```
vector<int> v = { 1, 3, 5, 6, 7 };
```

```
// Print first elt that is not odd
cout << *find_if_not
 (v.begin(),
 v.end(),
 [] (int i) {
 return i%2 == 1;
 });
```



# More algorithms

- partition\_copy

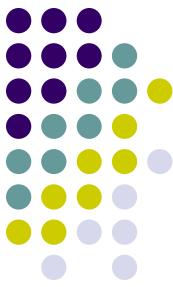
```
vector<int> primes;
vector<int> composites;
vector<int> data = getData();
extern bool is_prime(int i);
```

```
partition_copy
 (data.begin(),
 data.end(),
 back_inserter(primes),
 back_inserter(composites),
 is_prime);
```



# More algorithms

- `minmax`, `minmax_element`
  - Gets both the biggest and smallest items in the range
- `is_heap`, `is_heap_until`,  
`is_sorted`, `is_sorted_until`,  
`partial_copy`
- Ranged versions of move



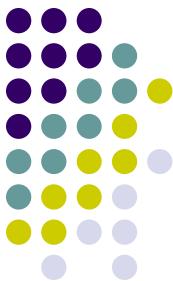
# Parallel algorithms

- While we haven't discussed threading yet, many standard library algorithms can run in parallel
- Just pass them a parallel execution policy as a first argument
- `sort(execution::par_unseq, v.begin(), v.end());`
- This does a parallel sort that can even (in principle) take advantage of GPUs
- <https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/>

# C++ algorithms seem pretty powerful, what's wrong?



- They have several problems that are becoming increasingly problematic as functional programming becomes more popular
- They can't be composed
  - In our transform then accumulate problem, we had to awkwardly create an (irrelevant) intermediate vector instead of just chaining
  - They can't work on infinite streams (it would take forever to create the intermediate vector)
  - The notation is cumbersome
    - Why do I have to say `sort(v.begin(), v.end())` instead of `sort(v)`?



# Introducing ranges

- Ranges are a work in process replacement for the C++ standard library algorithms that are
  - Composable so you can feed one algorithm into another
  - Work directly on ranges rather than iterator pairs for natural usage
  - Lazy for efficiency because you don't have to create intermediate containers holding all of the elements from each step
  - Fully conceptualized for improved safety and error messages
- Let's look at some example
  - <https://godbolt.org/z/PmtJRa>
  - <https://godbolt.org/z/RgHwaR>



# Ranges status

- A limited amount of ranges got into C++20
  - But not enough to be useful
  - Considered a sneak preview
  - Can't even do HW 2.1
- However, the open source implementation the standard is based on, Eric Niebler's Ranges v3 is very functional
  - Here is HW 2.1: <https://godbolt.org/z/hjaKf1>
  - Note how much better it is than HW 2.1
- C++23 should be good enough for it to be the main approach for algorithms (although it doesn't yet seem implemented)
- Warning: The final standard may differ in important ways from Ranges v3, so beware of technical debt



# BEST PRACTICES ON SPECIAL MEMBER FUNCTIONS



# Rule of three

- Even if you don't write them, the compiler automatically generates a
  - Copy constructor
    - `struct X { X(X const &); /* ... */ };  
X x2(x1);`
  - Copy assignment operator
    - `struct X { X &operator=(X const &); /* ... */ };  
x2 = x1;`
  - Destructor
    - `struct X { ~X(); /* ... */ };`
- Trick question: Which is called in the following
  - `X x2 = x1;`
  - Answer. Copy constructor! Although it looks like an assignment, it is constructing a new object, not copying over an existing object



# Rule of three (continued)

- Although the compiler generates those
- The built-in ones often aren't what you want
- This typically occurs when you need special copying behavior like a deep copy
  - Which usually means you need to do a deep copy-assignment
  - And a deep destruction



# Rule of three (continued)

- If your class defines any of the following
  - A copy constructor
  - A copy assignment operator
  - A destructor
- You should consider whether you need to define all three



# Rule of 3: example

- In our animal game example, our Node class is not copyable
  - Why?
- While we didn't need it, suppose we wanted to make it possible to copy a Node so we could "snapshot" a game
- Let's see the changes



# Rule of the Big Two

- In our example
- We didn't need to redefine the destructor because unique\_ptrs do all the work
- The rule of the Big Two says you don't need to worry about the destructor if the member destructors clean everything up
- We will generalize this later with the RAII principle

# Base classes should have virtual destructors



- We already discussed this, but including here for reference
- If you have a class that is designed to be inherited from
- Make sure its destructor is virtual
  - Since it might get destroyed by a `unique_ptr` to the base type
- ```
struct Animal {  
    /* ... */  
    virtual ~Animal() = default;  
};
```



TEMPLATES



Generic programming

- There are several ways to write code that can apply the same behavior across different types
- The most common are object-orientation and templates
- The terms template and generics are used almost synonymously in C++
- We covered virtual functions and classes last week
- This week, we'll look at templates



Simple function template

- We gave this example in Lecture 1

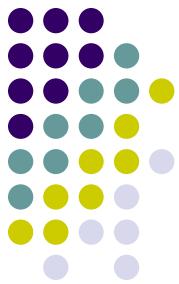
- auto square(auto x) {
 return x*x;
}
int i = square(21); // 441
double d = square(2.1); // 4.41

Is a function template a function?



- Not exactly
- It is a “template” that the compiler can use to create functions
- `square(21)` created a function `square<int>` from the template, compiles it and then builds a call
- `square(2.1)` is a call to `square<double>`
- If we later call `square(5)`, the compiler will observe that the function `square<int>` already was created and just builds a call to it

OO and Templates can solve similar problems



- Consider the following OO code

```
struct Animal {  
    virtual string name() = 0;  
    virtual string eats() = 0;  
};  
  
class Cat : public Animal {  
    string name() override { return "cat"; }  
    string eats() override { return "delicious mice"; }  
};  
// More animals...  
  
int main() {  
    unique_ptr<Animal> a = make_unique<Cat>();  
    cout << "A " << a->name() << " eats " << a->eats();  
}
```



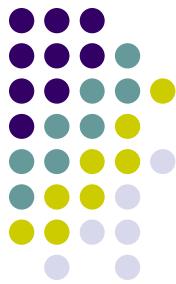
Do we need to use OO?

- Not really

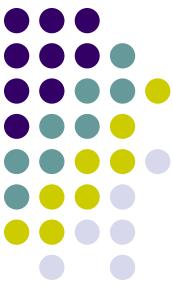
```
struct Cat {  
    string eats() { return "delicious mice"; }  
    string name() { return "cat"; }  
};  
// More animals...
```

```
int main() {  
    auto a = Cat();  
    cout << "A " << a.name() << " eats " << a.eats();  
}
```

That was a lot simpler but...



- We lost the understanding that `a` is an animal
- `a` could have the type `House` or `int` and we might not find out that something went wrong until much later when we did something that depends on `a` being an animal
- What we need is a way to codify our expectations for `a` without all of the overhead and complexity of creating a base class

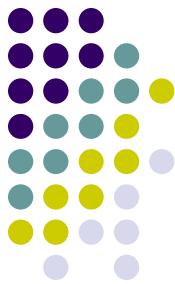


Concepts

- Concepts play the analogous role for generic programming that base classes do in object oriented programming
- A concept explains what operations a type supports
- The following concept encapsulates the same info as the base class

```
template<typename T>
concept Animal = requires(T a) {
    { a.eats() } -> convertible_to<string>;
    { a.name() } -> convertible_to<string>;
};
```

Now, we can ensure that a represents an animal



- With the above concept defined, we can specify that `a` must satisfy the `Animal` concept, and the compiler will not let us initialize it with a non-`Animal` type like `House` or `int`

```
int main() {  
    Animal auto a = Cat();  
    cout << "A " << a.name() << " eats " << a.eats();  
}
```



Let's compare

- <https://godbolt.org/z/cWc6aM>
- As you can see, the definition of Cat and the client code in main() look very similar in both
- This follows a principle enunciated by Bjarne Stroustrup
 - “Generic Programming should just be Normal Programming”



Kinds of templates

- Function templates, like `std::sort`
- Class templates, like `std::vector`
- Variable templates
 - Example of a pi template supporting different floating point types at
https://en.cppreference.com/w/cpp/language/variable_template



Matrix example

- To learn about class templates, we are going to create a "dense" matrix library
- This will also be a great opportunity to review the C++ features we have been learning and see how they combine to make an awesome library
- We are (very) loosely inspired by the matrix classes from the Origin C++ libraries, and lectures from Bjarne Stroustrup and Andrew Sutton

Sometimes you need a more verbose notation



- For function templates, we could simply use “concept auto” arguments
- If I have a class template, there are no “arguments,” so you put them up front in a “template” header
 - This is also useful if you need a name for the type
- You can use a concept
- `template<floating_point T> struct complex;`
- or if you don’t want to constrain, just say typename
 - This is all we had until C++20
- `template<typename T> // Simple vector def`
`struct vector { /* ... */};`



Matrices

- A matrix is just a two dimensional array of numbers (picture from Wikipedia)

$$\begin{bmatrix} 1 & 9 & -13 \\ 20 & 5 & -6 \end{bmatrix}.$$

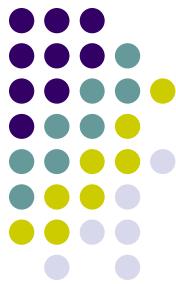
- Matrices are used in all branches of science, statistics, economics, math, etc. and can be added, multiplied, or have their determinants taken
- I will give you all necessary formulas



Initializing matrices

- We would like our Matrix class to have a natural initializer like the following 2x3 matrix
- $\text{Matrix}\langle 2, 3 \rangle \ m = \{ \{ 2, 4, 6 \}, \{ 1, 3, 5 \} \};$

Review: Initializer lists



- In C++98, one of the problems with using vectors was that it was difficult to initialize them

- // Yuck!

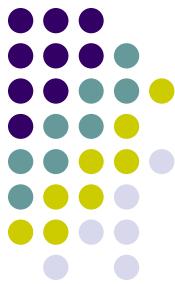
```
int init[] = { 0, 1, 1, 2, 3, 5 };
vector<int> v(init,
init+sizeof(init)/sizeof(init[0]));
```

Can't user-defined types be as easy to init as built-in ones?

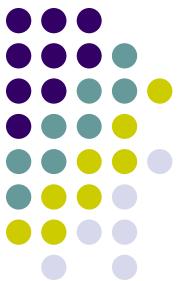


- In C++11, you can initialize vectors as easily as C arrays
 - `vector<int> v = {0, 1, 1, 2, 3, 5};`
 - `vector v = { 1, 2, 3, 4};`
- How does `vector<T>` do this?
- It has a constructor that takes a `std::initializer_list<T>`, which represents a “braced initializer of Ts” expression
- Initializer lists have `begin()`, `end()`, and `size()` methods so your constructor can iterate through their value.

Initializer list constructor for matrix



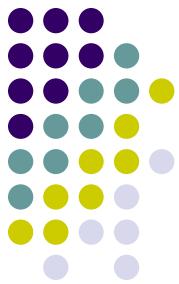
- The code to initialize as mentioned previously is on Canvas
- Note from the slide above that we initialize matrices with “initializer lists of initializer lists”
 - The initializer list contains an initializer list for each row
- ```
Matrix(initializer_list<initializer_list<double>> init) {
 auto dp = data.begin();
 for (auto row : init) {
 std::copy(row.begin(), row.end(), dp->begin());
 dp++;
 }
}
```



# Matrix arithmetic

- In order to easily add and multiply matrices, we would like to be able to tell “+” and “\*” about matrices
- This is called operator overloading

# Overloading operators for matrices



- The sample Matrix.h on Canvas overloads matrix multiplication with the (complicated) rule for multiplying matrices

# Specializing and overloading templates



- The “secret sauce” for C++ templates is that if the general “generic” definition of the template isn’t really what you want for a particular set of template parameters, you can override it for that particular case with a specialization
- Think of this as the compile-time analog to object orientation where you also override a more general method in a more specialized derived class



# Matrix determinants

- The determinant is a number that represents “how much a matrix transformation expands its input”
  - Don’t worry if you don’t understand this
- We will just use the formula to calculate them
- The general formula is here
  - [http://en.wikipedia.org/wiki/Laplace\\_expansion](http://en.wikipedia.org/wiki/Laplace_expansion)
- But I will give you the special case you need for the homework



# Full specialization

- A function, class, or member can be fully specialized
- See the definition of `Matrix<1,1>::determinant()` in `Matrix.h`



# Overloading

- Function templates can be overloaded
- For example, see OverloadMatrix.h



# Partial specialization

- Only classes may be partially specialized
- Template class:

```
template<class T, class U>
class Foo { ... };
```

- Partial specialization:

```
template<class T>
class Foo<T, int> { ... };
```

- You can tell the second is a specialization because of the <> after the class name



# Partial specialization

- The partially specialized class has no particular relation to the general template class
  - In particular, you need to either redefine (bad) or inherit (good) common functionality
  - For example, see PSMatrix.h



# Exercise 4-1

- Modify Matrix.h to let you add matrices
  - Use operator overloading to make both + and += work
- To add two matrices, they both have to have the same number of rows and columns
- Just add the corresponding elements to get a new matrix with the same number of rows and columns
- See  
<http://www.purplemath.com/modules/mtrxadd.htm> for an example



## Exercise 4-2

- For each of the following programs, modify them to have a direct (i.e., specialized or overloaded implementation) of determinants for  $2 \times 2$  matrices.
  - Matrix
  - PSMatrix
  - OverloadMatrix
- The formula for the determinant of the  $2 \times 2$  matrix  $m$  is
$$m(0, 0) * m(1, 1) - m(1, 0) * m(0, 1)$$
- Test how much your code changed the execution time for the programs. What do you conclude?



## HW 4-3

- This problem digs into the challenges of passing arguments to template functions, not unlike the ones we will encounter in passing arguments to thread functions
- The following code (next slide) to find the maximum length of a collection of strings unexpectedly always prints 0. Why doesn't it work? How can you fix it?
  - Looking up the documentation of `for_each` may suggest a possible solution



# HW 4-3 (Code)

```
#include<algorithm>
#include<iostream>
#include<string>
#include<vector>
using namespace std;

struct maxlenftn {
 maxlenftn() { maxlen = 0; }
 void operator()(string s) {
 maxlen = max(maxlen,s.size());
 }
 string::size_type maxlen;
};

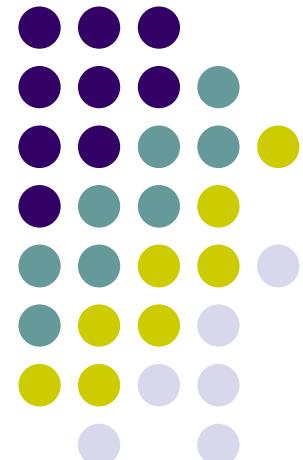
int main() {
 vector<string> names{"Smith", "Jones", "Johnson", "Grant"};
 maxlenftn maxf;
 for_each(names.begin(),names.end(),maxf);
 cout << maxf.maxlen << endl;
 return 0;
}
```

# C++

## February 2, 2023

Mike Spertus

[spertus@uchicago.edu](mailto:spertus@uchicago.edu)



MASTERS PROGRAM IN  
COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO

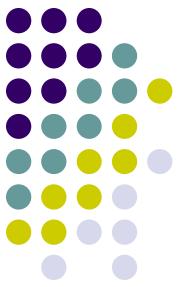
# Compilation of template methods



- You might wonder why `Matrix<int, 1, 1>` objects can be created even though `Matrix<int, 1, 1>::minor(int, int)` doesn't make sense (it would be a 0x0 matrix)
- Answer: If a method of a template class isn't called, then it isn't compiled
- Really helps us here, but sometimes you might wonder why, say, a static member is never compiled

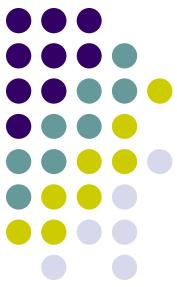


# MAKING MATRIX BETTER



# static\_assert

- What happens if we call determinant() on a non-square matrix?
- Let's give it a try
- Wow! That was a weird error message
- Can we do better?



# static\_assert

- C++ lets you create a compile-time assertion that prints a nice error message of your choice
- `static_assert(rows == cols,  
 "Sorry, only square matrices have determinants");`
- Much better



# \*this

- Sometimes a method needs a reference to the object it is a part of
  - `*this`
- In `OverloadMatrix`, `Matrix::determinant()` uses `this` to pass the containing `Matrix` to the external `determinantImpl` function

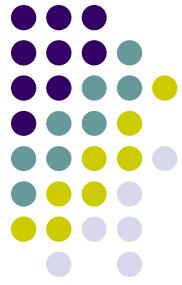
```
template<class T, int h, int w>
T
Matrix<T, h, w>::determinant() const
{
 return determinantImpl(*this);
}
```

- This also comes in handy in the homework for overloading `Matrix::operator+=`



# Inheriting constructors

- It seems annoying that in `PSMatrix`, we can inherit everything but constructors from the common implementation even though they do the right thing
  - Indeed, the constructors don't even have the same names, because they are named after the class
  - Since classes often have many constructors, this can make it painful to build lightweight wrapper classes on top of others
- We can solve this by saying we want to inherit constructors
- ```
class Matrix : public MatrixCommon < T, rows, cols > {  
public:  
    Matrix() = default;  
    Matrix(initializer_list<initializer_list<T>> init) :  
        MatrixCommon<T, rows, cols>(init) {}  
using MatrixCommon<T, rows, cols>::MatrixCommon;
```



TEMPLATES BEFORE CONCEPTS

Wait, Concepts? I'm not sure I remember what they are...



- Besides auto, the other major C++20 template feature is Concepts, which us allow us to constrain our templates
- `template<std::floating_point T, int m, int n = m>`
`struct Matrix { /* ... */ };`



Another example

- The algorithms we've been discussing often require a callable
- How would we write a function template that expects its argument is callable with an int?
- ```
void f(std::invocable<int> auto callable) {
 callable(5); // https://godbolt.org/z/M8axbv7Ys
}
```
- What if we just didn't use a Concept?
- ```
void f(auto callable) {
    callable(5); // https://godbolt.org/z/M8axbv7Ys
}
```
- It still works. It runs with a valid callable and fails to compile if callable cannot be called with an int
- However, the error will be inside the body of f where the call takes place
- In this case, it's not a big deal since f is so simple
- But let's look at some real-life examples where it really helps to constrain the template



Here's another example

- What goes wrong with the following natural code?

```
template<typename T>
void f(T &t) {
    sort(t.begin(), t.end()); // OK
}
vector<int> v = { 3, 1, 4, 1};
list<int> l = {5, 9, 2};
f(v); // OK
f(l); // Huge compiler error
```



What went wrong?

- sort requires random access iterators
- Since lists need to be walked through a node at a time, their iterators are not considered random access
- Somewhere deep inside of the sort algorithm, something goes wrong, leading to an incomprehensible error message
 - Might be hundreds of lines long

You can't overstate what a big problem this is



- As we've discussed numerous times, C++ is a language of lightweight abstractions
- The theory of templates is to provide the ideal abstract interface at zero cost as a template library
- In practice, that works well if your client uses your templates correctly
- But any mistake leads to incomprehensible error messages



Easy to solve with Concepts

- `template<random_access_iterator it>`
`sort(it beg, it end);`
- Now the error would be something understandable like
 - “Sorry, `list::iterator` does not satisfy the `random_access_iterator` concept”

Even worse, it may run but give the wrong answer



- Consider this implementation of the Greatest Common Divisor function based on Euclidean Algorithm
- ```
int gcd(int a, int b) {
 if(b == 0)
 return a;
 return gcd(b, a - b*(a/b));
}
```
- This efficiently and correctly calculates greatest common divisors. E.g., `gcd(48, 30)` returns 6

# But what about types other than ints



- Of course, we may want to take greatest common divisors of other types, like unsigned, long, etc.
- Indeed, this is the heart of RSA cryptography
- Do we need to write a million gcd functions?
- Of course not, that's what templates are for



# GCD template

- ```
template<typename T>
T gcd(T a, T b) {
    if(b == 0)
        return a;
    return gcd(b, a - b*(a/b));
}
```
- This correctly calculates gcd's like `gcd(48, 30)`, `gcd(48u, 30u)`, `gcd(481, 301)`, etc.



So what's the problem?

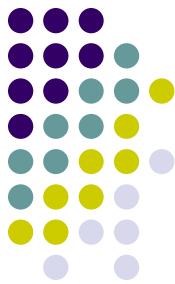
- Lack of constraints on the parameter!
- This function template can easily be instantiated with types that don't satisfy the assumptions on the parameters
- $\text{gcd}(4.0, 6.0)$ compiles but crashes at runtime
 - Infinite recursion due to template rounding
 - <https://godbolt.org/z/fWEfPx1ad>
- Worse yet, $\text{gcd}(4.0, 8.0)$ runs but gives the incorrect answer of 4.0. (Any double divides any other double, so it makes no sense to call that the Greatest Common Divisor)



Lesson from history

- In the original K&R version of C, function prototypes didn't need argument lists
 - `int gcd();`
- Likewise, it was easy to inadvertently call with inappropriate arguments
 - `gcd(12l, 16l); // crashed!`
- The solution was to require prototypes in declarations to ensure valid arguments
 - `int gcd(int, int);`

What we need is prototypes for templates

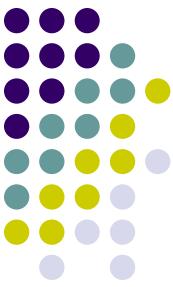


- In C++20, we can give an analogous “prototype” to our template
- ```
template<integral T>
T gcd(T a, T b) {
 if(b == 0)
 return a;
 return gcd(b, a - b*(a/b));
}
```
- “Generic programming should be like normal programming” --- Bjarne Stroustrup
- But what do we do in Classic C++?



# SFINAE

- “Substitution Failure is Not an Error”
- When creating candidate overloads for a function, invalid substitutions in a particular function template mean that template is not a possible overload
- But no compile error occurs even though the template failed to compile



# Example (Wikipedia)

```
struct Test {
 typedef int foo;
};
template <typename T>
void f(typename T::foo) {} // Definition #1
template <typename T> void f(T) {} // Definition #2

int main() {
 f<Test>(10); // Call #1.
 f<int>(10); // Call #2.
 // Without error (even though there is no int::foo)
 // thanks to SFINAE.
}
```



# std::enable\_if

```
template <bool B, class T = void>
struct enable_if {
 typedef T type;
};
```

```
template <class T>
struct enable_if<false, T> {};
```

```
template<bool B, typename T>
using enable_if_t = enable_if<B, T>::type;
```

# Using SFINAE to fix our gcd function template



- We only want gcd to be used with integral types
- The standard library has an `is_integral_v` template that checks if a type is integral
- `is_integral_v<unsigned>` is true
- `is_integral_v<double>` is false
- Only allow gcd to be called on integral types

```
template <class T, typename = enable_if_t<is_integral_v<T>>>
T gcd(T a, T b) {
 if(b == 0) return a;
 return gcd(b, a - b*(a/b));
}
```



# What's the point?

- `enable_if<b, T>::type` is the type `T` if `b` is true
- `enable_if<b, T>::type` is a substitution failure if `b` is false
- Use it in template parameter lists, return values or argument types to suppress or enable the generation of certain template functions by SFINAE
- `gcd(4, 6); // OK`
- `gcd(4.2, 6.0); // Must be a different gcd`



# CONSTEXPR



# constexpr allows you to do things at compile-time

- Normally we think of programming as a way to write code that runs at runtime
- It is surprisingly common that you need “something” to take place at compile-time instead of runtime
- What do I mean by “something”?
- We will look at several examples over the next few slides

# Template arguments need to be known at compile time



- The following code doesn't compile
  - ```
int n;
cout << "How big a matrix? ";
cin >> n;
Matrix<n, n> m; // Ill-formed!
```
- The point: Since we don't know n at compile-time, the compiler can't compile the class `Matrix<n, n>`



Does this compile?

- Consider the following variation
- ```
auto square(int x) { return x*x; }
Matrix<square(3), square(3)> m;
```

# Perhaps, surprisingly, it does not?



- While we can see from looking at the code that `square(3)` is going to be 9
- The compiler can't make that assumption
- Imagine what would go wrong if it did
  - Whether the code is legal would depend on whether the optimizer ran `square` at compile time or run-time
    - Which would just be weird
  - Also, if someone changed the body of `square`, it might no longer be computable at compile-time
- Therefore, the compiler has to reject `square(3)` as a template argument just like it did `n` in the previous example

# Doing things at compile-time can also help performance



- Not only do you need compile-time values for template arguments
- But they benefit performance as well
- Why compute the following every time the program is run?

```
double pi = 4 * atan(1);
```



# Constant expressions

- A *constant expression* is an expression that can be evaluated at compile-time
- Built-in values like `3`, `2 * 7` and `false` are constant expressions
- But since templates and performance programming are so important in C++, we would like to be able to create our own constant expressions
- That is what `constexpr` does
- Let's look at some examples



# constexpr variables

- We've already seen constexpr variables

```
int constexpr seven = 7; // immutable
```
- This means that wherever the compiler sees seven, it can substitute the constant expression 7



# constexpr functions

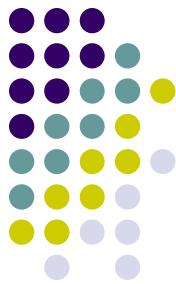
- Consider

```
double const pi = 4*atan(1);
```

- Wouldn't it be nice if that could be calculated at compile-time?
- To say that a function is computable at compile-time, label it as `constexpr`
- Example: Calculate greatest common divisor by Euclidean Algorithm

- ```
int constexpr gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

When do `constexpr` functions run?



- If the arguments are known at compile-time, the function may be run either at compile-time or at run time at the compiler's discretion. E.g., `gcd(34, 55)`
- If the value is needed at compile-time, it is calculated by the compiler. E.g., as a non-type template parameter: `Matrix<gcd(34, 55)>`
- If its arguments are not known at compile-time, it won't be run until run-time
 - ```
void f(int i) {
 return gcd(34, i); // i is unknown
}
```



# constexpr functions

- A constexpr function cannot contain just any code.
  - For example, if it contained a thread\_local variable, what would that mean at compile-time?
- What is not allowed:
  - Uninitialized declarations
  - static or thread\_local declarations
  - Modification of objects that were not created in the function
    - Or potential modifications by, say, calling a non-constexpr function.
  - virtual methods
  - Non-literal return types or parameters
    - A type is literal if its constructor is trivial or constexpr

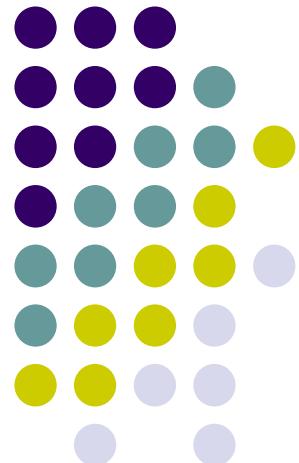


# Fixing the square example

- Now our code runs
- ```
auto square(int x) constexpr
{ return x*x; }
```

```
Matrix<square(3), square(3)> m;
```

if constexpr





The problem with if

- All of our Matrix examples may have seemed like overkill
- If we really wanted a different determinant method for 1×1 matrices, how come we didn't just use if to check if it is a 1×1 matrix and then use the right formula?

Trying to find a determinant with if



```
template<int r, int c = r>
class Matrix {
    /* ... */
    double determinant() {
        if(r == 1 && c == 1) // 1x1 Matrix
            return data[0][0];
        else {
            double val = 0; // Bigger. Use formula
            for (int i = 0; i < r; i++) {
                val += (i % 2 ? -1 : 1) * data[i][0]
                    * minor(i, 0).determinant();
            }
        }
        return val;
    }
}
```

Unfortunately, that code won't compile on a 1×1 matrix



- The problem is that the else branch takes a minor of a 1×1 matrix, which doesn't make sense (it would be a 0×0 matrix)
- Wait a moment! If the matrix is 1×1 , the else branch will never be run, so who cares?



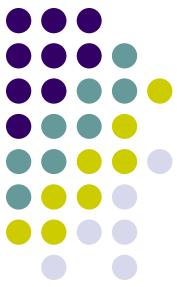
The problem with if (cont)

- Even though the else branch of the if will not run on a 1x1 matrix (since the condition is guaranteed to be true), the compiler still needs to compile it
- But `Matrix<1, 1>::minor(i, 0)` will be illegal gobbledegook
- So the compilation fails



The problem with if (cont)

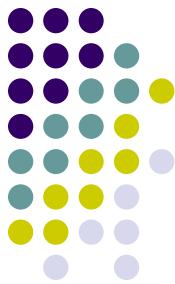
- This seems really restrictive
- Since the compiler can tell the else branch of the if won't be run, why can't it just ignore the illegal code inside it?
- That's where `if constexpr` comes in



if constexpr

- Saying `if constexpr` rather than just `if` tells the compiler not to compile the branch not run if the condition is known at compile-time
- Just replacing the `if` with `if constexpr` makes it work

Successfully finding a determinant with if constexpr



```
// No specialization or overloading required

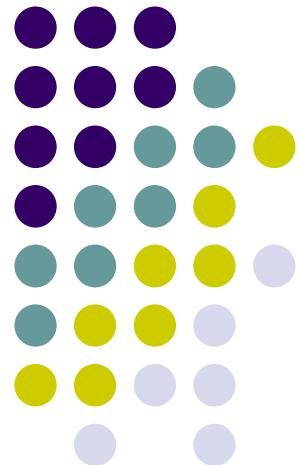
template<int r, int c = r>
class Matrix {
    /* ... */
    double determinant() {
        if constexpr (r == 1 && c == 1) // 1x1 Matrix
            return data[0][0];
        else {
            double val = 0; // Bigger. Use formula
            for (int i = 0; i < r; i++) {
                val += (i % 2 ? -1 : 1) * data[i][0]
                    * minor(i, 0).determinant();
            }
        }
        return val;
    }
}
```



const vs constexpr

- It is easy to confuse const and constexpr, but they mean different things
- constexpr means the value is known at compile-time
 - `int constexpr seven = 7; // seven means 7`
- const means the value won't be changed
`void f(int const &i);`
- We don't know the value of i until runtime, but we know f won't change it
 - Actually, it is possible for f to change i (but it shouldn't, so I will ignore it for now)

decltype

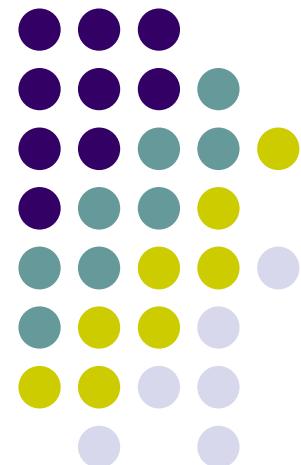


Sometimes you want to know the type of a value



- Suppose I don't know what type an expression x is
 - Very possible in template code or with complex expressions (e.g., the type of $1u + 'c'$ is implementation-defined depending on whether `char` is signed or unsigned)
- `decltype(x)` gives you the type of x
- `Matrix<decltype(x), 1, 2> m = { {x, x} };`

Move Semantics





How to pass an argument

- Pass by value

void f(X x);

- Pass by reference

void f(X &x);

- Pass by move

void f(X &&x); // What we will learn

- Most languages only have one choice

- C passes by value, Java by reference, and Rust by move

- You will not be surprised to find out that C++ supports all 3

What does it mean to move an object



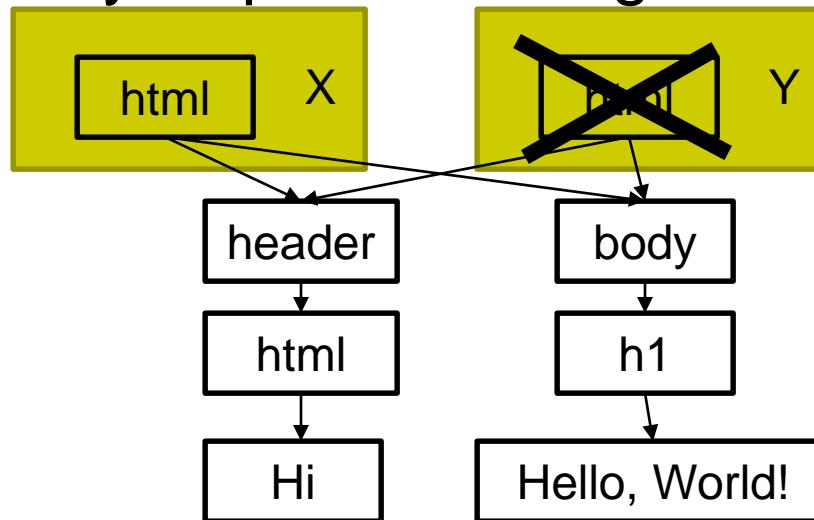
- Moving y to x
 - $x = \text{move}(y);$
- Means that the value of x is the same as the value y started at
- So one way to do this is by using ordinary copying like
 - $x = y;$
- However, the difference between a copy and a move is that y is allowed to be changed by the move
 - Subsequent code should not use the value of y

Why move rather than copy?

Efficiency

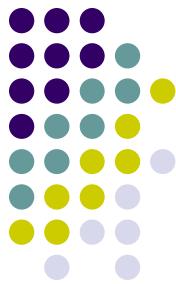


- Since a move can “raid the old object for parts,” it can be a lot more efficient
 - Copying a tree requires copying all of the branches
 - Moving a tree only requires moving the root



Why move rather than copy?

Semantics



- Some classes don't make sense to copy and therefore have deleted copy constructors
 - ```
struct X {
 X(X const &) = delete;
 /* ... */
};
```
- E.g., if you copied a unique\_ptr, the managed object would be owned by two unique\_ptrs
  - Oops!
- Better to use a move

# How does C++ know whether to move or copy?



- If it is sure that the old object doesn't need its value any more
- One way is if the old object is a temporary value and has no name, so it can't be referred to again
  - In the following example, it is safe to move the unique\_ptr returned by make\_unique into x, since that is the only place the return value is visible
    - `auto x = make_unique<X>(); // x owns the X obj`
- The other way is if the programmer uses `std::move` to say that it is ok to move from
  - `auto y = move(x); // Now y owns the X obj`

# How does move work?

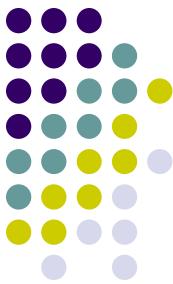
## Rvalue references



- A reference with `&&` instead of just `&` can bind to a temporary and move it elsewhere.
- Objects are often much cheaper to “move” than copy
  - For example, deep copying a tree
  - Some objects, like `unique_ptr` can be moved but not copied
- template<class T>

```
void swap(T& a, T& b)// "perfect swap"(almost)
{
 T tmp = move(a); // could invalidate a
 a = move(b); // could invalidate b
 b = move(tmp); // could invalidate tmp
}
```

# Move semantics example: putting items into an vector



- Recall that `std::unique_ptr` is not copyable
- Since they are movable, we can construct a temporary `unique_ptr` and move it into a vector
- ```
template<typename T> class vector {  
    ...  
    push_back(T const &t);  
    push_back(T &&t);  
    ...  
};
```
- ```
vector<unique_ptr<btree>> vt;
for(int i = 0; i < 10; i++) {
 vt.push_back(make_unique<btree>());
}
```



# “Rvalue reference references”

- Here are some useful references on rvalue references
- [http://thbecker.net/articles/rvalue\\_references/section\\_01.html](http://thbecker.net/articles/rvalue_references/section_01.html)
  - What I lectured from in class
- <https://medium.com/pranayaggarwal25/move-semantics-269e73287b63>
- A whole book on all about rvalue references
  - Nico Josuttis, *C++ Move Semantics: The Complete Guide* - <http://www.cppmove.com/>
  - Nico guest lectured on Move Semantics last spring

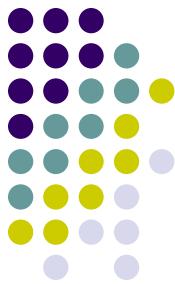
# How do I make a type movable?



- First, you don't *need* to make a class movable. It is only a performance optimization because C++ will just copy if there are no move operations
- If moving is more efficient, you should create move constructors and move assignment operators, just like standard library containers do:
- ```
template<class T> class vector {  
    // ...  
    vector(vector<T> const &); // copy constructor  
    vector(vector<T> &&); // move constructor  
    vector& operator=(const vector<T>&); // copy assignment  
    vector& operator=(vector<T>&&); // move assignment };
```
- Sometimes, the compiler will automatically generate move constructors and move assignment operators that just move all the members
 - Basically if you don't define a copy constructor/assignment operator or a destructor
- If you want to force the compiler to generate the default move constructor even though it wouldn't normally, you can force that with `default`
- ```
struct S {
 S(S const &); // OK, but stops move constructor generation
 S(S &&) = default; // Gets it back
 /* ... */
};
```

# Summary

## The three ways of passing



- By value
  - The function gets its own copy
    - // Doesn't change callers int  
`void f(int i) { i += 2; }`
- By (lvalue) reference
  - The function works with the caller's object
  - // Changes callers int  
`void f(int &i) { i += 2; }`
- By rvalue reference
  - If the function is called on a temporary
  - // Can reuse parts from abandoned x  
`void f(X &&x) { ... }`



# Rule of five?

- There is a lot of discussion about whether the rule of 3 should be extended to a “rule of 5,”
  - If you define any of
    - The destructor
    - The copy constructor
    - Copy assignment operator
    - Move constructor
    - Move assignment operator
  - You should review that they all do the right thing because they are all related
- C++11 deprecated some features to better mesh with the rule of 5
  - A proposal (with history) to remove the deprecated features was rejected for C++14. Even though it was rejected it makes interesting and illuminating reading for aspiring language lawyers
    - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3578.pdf>
    - Warning: This paper is hard-core. Only recommended if you have substantial C++ experience

# How is std::move implemented? (Very advanced)



- Yet another “way of argument passing”
- If `&&` is used in a template argument, it means “forwarding reference,” which bind to either an rvalue reference or a regular (lvalue) reference
- Template type deduction takes place according to the following “collapsing rules” which are only applied in templates
  - $T\& \ \& \cong T\&$
  - $T\& \ \&\& \cong T\&$
  - $T\&\& \ \& \cong T\&$
  - $T\&\& \ \&\& \cong T\&\&$



# std::move code

- template <class T>  
`remove_reference_t<T> &&`  
`move(T&& a)`  
`{ return a; }`
- What happens in the code  
`A a;`  
`f(move(a));`
- For what T is `T&&` an `A` or `A&`?
- By the collapsing rules, we see that the only option is that  
 $T \cong A\&.$  ( $T \&\& \cong A\& \&\& \cong A\&$ )
- Now, we return a  
`remove_reference_t<A&>&& \cong A\&&`
- Which tells f that a can be “raided for parts.”
- If you are interested, you can check that all the other cases work



# Perfect forwarding

- A lot of times you want to wrap a function and pass your arguments to it
  - “Every problem in programming can be solved with another layer of indirection”
- This is surprisingly difficult because you have to properly forward all the different ways of passing parameters (e.g., lvalue references vs rvalue references)
- We will consider our own implementation of `make_unique` based on  
[http://thbecker.net/articles/rvalue\\_references/section\\_07.html](http://thbecker.net/articles/rvalue_references/section_07.html)
  - Patterns like this that create objects are known as “factory patterns.”

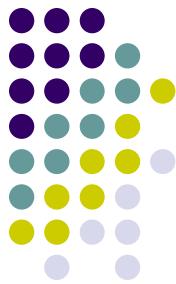
# A naïve make\_unique implementation



- A first stab at make\_unique would look something like
- ```
template<typename T, typename Arg>
unique_ptr<T> makeUnique(Arg arg)
{
    return unique_ptr<T>(new T(arg));
}
```
- This code assumes that T takes one constructor argument
- But of course it might take more or fewer

Our first improvement

Variadics

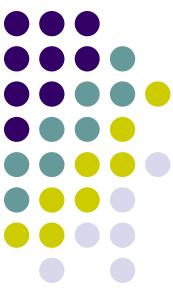


- C++ lets you use ... to indicate a “pack” of template parameters
- While there are a lot of details, the basic use is straightforward
- ```
template<typename T, typename... Arg>
unique_ptr<T> makeUnique(Arg... arg)
{
 return unique_ptr<T>(new T(arg...));
}
```

# Looks alright, so what's wrong?



- ```
struct X {  
    // This constructor modifies its argument  
    X(int &i) { i++; }  
};
```
- Oops, modifies a copy of its argument
- ```
int i = 2;
auto xp = makeUnique<X>(i); // i is still 2
```
- The point is that `makeUnique` infers that `Args...` is `int`, so `i` is passed by value.



# Maybe if we took our arguments by reference?

- ```
template<typename T, typename... Args>
unique_ptr<T> makeUnique(Args &... args)
{
    return unique_ptr<T>(new T(args ...));
}
```
- Now, this works
- ```
int i = 2;
auto xp = makeUnique<X>(i); // i is 3
```



# Not so fast

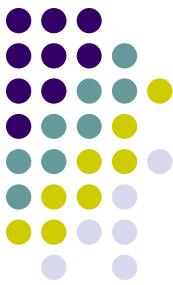
- Can't bind a T & to an rvalue, so if I call `make_unique` with any rvalue arguments then I will get a compile error
- ```
struct X { X(int const &); };
new X(2); // Legal
// We would like the following line
// to also be able to create an X(2), but..
makeUnique<X>(2); // Ill-formed!
```
- What went wrong?
- Our (current) argument list for `make_unique` tries to bind to 2 with an `int &`, but it obviously doesn't make sense to refer to the constant numerical value 2 with an `int &` because such a non-const reference could try to change the value of 2!

This gets to be an ugly mess real fast



- We haven't even considered if `x` has a move constructor yet
- We might be able to get around this by creating a zillion overloads, but thinking about the case of `x` having a constructor that takes some arguments by value, some by reference, some by const reference, and some by rvalue reference means that `makeUnique` would really need a zillion overloads

The perfect-forwarding problem



- How can we pass our arguments as-is to another function using that function's signature
- The standard supplies a `std::forward` function that does exactly that
- ```
// Perfectly forwards everything
template<typename T, typename... Args>
unique_ptr<T> makeUnique(Args &&... args)
{
 return unique_ptr<T>(new T(forward<Args>(args)...));
}
```



# How does std::forward work?

- You definitely don't need to understand this (or most of the preceding slides) to use std::forward
- But if you are interested, it consists of two function templates
  - ```
template< class T >
constexpr T&& forward(remove_reference_t<T>& t )
{ return t; }

template< class T >
constexpr T&& forward(remove_reference_t<T>&& t )
{ return t; }
```
- For example, `forward<int const &>(7)` leverages the template collapsing rules to forward it as an
 $\text{int const \& \&\&} \cong \text{int const \&}$
as desired



THREADS

Computers are not getting faster



- Perhaps the biggest secret in computer progress is that computer cores have not gotten any faster in 15 years
 - 2005's Pentium 4 HT 571 ran at 3.8GHz, which is better than many high-end CPUs today
 - The problem with increasing clock speeds is heat
 - A high end CPU dissipates over 100 watts in about 1 cubic centimeter
 - An incandescent light bulb dissipates 100 watts in about 75 cubic centimeters



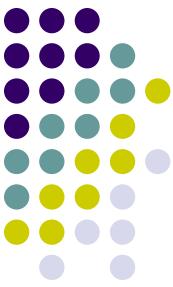
Computers are faster

- Even though cores have not gotten faster, the continued progression of Moore's law means that computers today have many cores to run computations in parallel
 - Even cell phones can have 4 cores
 - 12 to 24 cores are not unusual on high-end workstations and servers
 - 24 to 48 if you count hyperthreading



What does this mean to me?

- If you don't want your code to run like it's 2005, you need your code to run in parallel across multiple cores
- In other words, **you need threads!**



Hello, threads

```
#include <iostream>
#include <thread>

void hello_threads() {
    std::cout<<"Hello Concurrent World\n";
}

int
main(){
    // Print in a different thread
    std::thread t(hello_threads);
    t.join(); // Wait for that thread to complete
}
```



What happened?

- Constructing an object of type `std::thread` immediately launches a new thread, running the function given as a constructor argument (in this case, `hello_threads`).
 - We'll talk about passing arguments to the thread function in a bit.
- Joining on the thread, waits until the thread completes
 - Be sure to join all of your threads before ending the program
 - Exception: Later we will discuss detached threads, which don't need to be joined



Locks

- The simplest way to protect shared data is with a `std::mutex`
- Accessing the same data from multiple threads without using synchronization like mutexes is called a data race error
 - Don't do this!
- How can we make sure we release the mutex when we are done no matter what?
- RAII!
- C++11 includes a handy RAII class `std::lock_guard` for just this purpose.
- C++17 adds `std::scoped_lock` that can lock any number of locks **without deadlocking!**



Locks

```
std::list<int> some_list; // A data structure accessed by multiple threads
std::mutex some_mutex; // This lock prevents concurrent access to the shared data structure

void
add_to_list(int new_value)
{
    // Since I am going to access the shared data struct, acquire the lock
    std::lock_guard guard(some_mutex); // CTAD deduces lock_guard<mutex>
    some_list.push_back(new_value);
    // Now it is safe to use some_list. lock_guard destructor releases lock at end of function
    /* ... */
}

bool
list_contains(int value_to_find)
{
    std::lock_guard guard(some_mutex); // Must lock to access some_list
    return std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
}
```

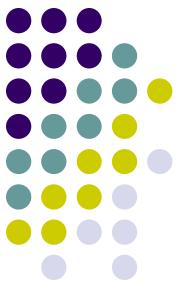


shared_mutex

- A shared_mutex can be acquired either in shared ownership mode or unique ownership mode
 - ```
shared_mutex sm;
shared_lock sl(sm);
// RAII for acquiring sole ownership
unique_lock ul(sm);
```
- Motivating use case: reader-writer lock
  - Multiple threads can read from a data structure at the same time as long as no thread is modifying it
  - If a thread is modifying the data structure, it should acquire sole ownership of the object so no thread sees the data structure in a partially modified state

# Reader-Writer Locks

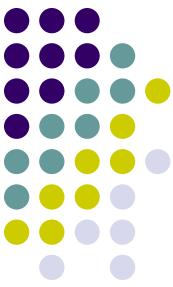
## shared\_mutex



```
std::list<int> some_list;
std::shared_mutex some_mutex;

void
add_to_list(int new_value)
{
 std::unique_lock guard(some_mutex); // Unique writer access
 some_list.push_back(new_value);
}

bool
list_contains(int value_to_find)
{
 std::shared_lock guard(some_mutex); // Shared reader access
 return
 std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
}
```



# Lock ordering

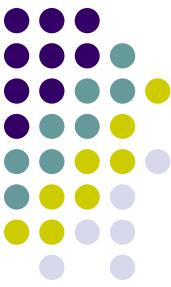
- If you want to avoid deadlocks, you want to acquire locks in the same order!
  - Suppose thread 1 acquires lock A and then lock B
  - Suppose thread 2 acquires lock B and then lock A
  - There is a window where we could deadlock with thread 1 owning lock A and waiting for lock B while thread 2 owns lock B and is waiting for lock A forever
- The usual best practice is to document an order on your locks and always acquire them consistent with that order
- See <http://www.ddj.com/hpc-high-performance-computing/204801163>
- Tools like ThreadSanitizer can also help find potential deadlocks and data races
  - <https://clang.llvm.org/docs/ThreadSanitizer.html>

# Sometimes it is hard to fix a lock order



- From <http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-7-locking-multiple-mutexes.html>
- Consider

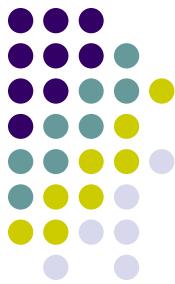
```
class account {
 mutex m;
 currency_value balance;
public:
 friend void transfer(account& from,account& to,
 currency_value amount) {
 lock_guard lock_from(from.m);
 lock_guard lock_to(to.m);
 from.balance -= amount;
 to.balance += amount;
 }
};
```
- If one thread transfers from account A to account B at the same time as another thread is transferring from account B to account A: Deadlock!



# C++ provides a solution

- `std::scoped_lock` allows you to acquire multiple locks “at the same time” and guarantees there will be no deadlock,
  - Like magic! (Actually, it will try releasing locks and then acquiring in different orders until no deadlock occurs)
- ```
class account {  
    mutex m;  
    currency_value balance;  
public:  
    friend void transfer(account& from,account& to,  
                         currency_value amount) {  
        scoped_lock lck(from.m, to.m);  
        from.balance -= amount;  
        to.balance += amount;  
    }  
};
```

Not so basic: Thread arguments



- You can add arguments to be passed to the new thread when you construct the `std::thread` object as in the next slide
- But there are some surprising and important gotchas that make passing arguments to thread function different from passing arguments to ordinary functions, so read on



Passing arguments to a thread

```
mutex io_mutex;

void hello(string name) {
    lock_guard<mutex> guard(io_mutex);
    cout <<"Hello, " << name << endl;
}

int
main(){ // No parens after thread function name:
    vector<string> names = { "John", "Paul"};
    vector<thread> threads;
    for(auto name : names) {
        threads.push_back(thread(hello, name));
    }
    for(auto it = threads.begin(), it != threads.end(); it++) {
        it->join();
    }
}
```



Deceptively simple

- A different notation is used from arbitrary function calls, but otherwise fairly straightforward looking:
 - `void f(int i);`
`f(7); // Ordinary call`
`thread(f, 7); // f used as a thread function`

Gotcha: Signatures of thread functions silently “change”



- What does the following print?

```
void f(int &i) { i = 5; }

int main() {
    int i = 2;
    std::thread t(f, i);
    t.join();
    cout << i << endl;
    return 0;
}
```



A compile error

- Of course, 5 was intended
- Unfortunately, thread arguments are not interpreted exactly the same way as just calling the thread function with the same arguments
- This means that even an application programmer using threads needs to understand something subtle about templates



What went wrong, continued

- thread's constructor looks like the following

```
struct thread { ...  
    template<typename func, typename... arg>  
    thread(func f, arg... a);  
    ...  
};  
...  
// Deduces thread::thread<void(*)(int&), int>  
std::thread t(f, i);  
...
```



IOW, Templates don't know if takes its argument by reference

- To do this, we will use the “ref” wrapper in `<functional>`
- ```
void f(int &i) { i = 5; }

int main() {
 int i = 2;
 std::thread t(f, std::ref(i));
 t.join();
 cout << i << endl;
 return 0;
}
```



# Gotcha: Passing references

- Be very careful about passing pointers or references to local variables into thread functions unless you are sure the local variables won't go away during thread execution

- Example (based on Boehm)

```
void h(int &i);
void f() {
 int i;
 thread t(h, ref(i));
 bar(); // What if bar throws an exception?
 t.join(); // This join is skipped
} // h keeps running with a pointer
// to a variable that no longer exists
// Undefined (but certainly bad) behavior
```

- Use try/catch or better yet, a RAII class that joins like the jthread



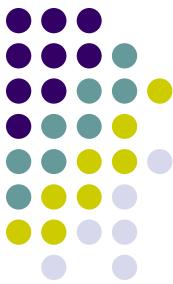
# EXCEPTIONS AND RAI



# Exceptions

- When something goes wrong, throw an exception
  - This ensures that errors are never inadvertently ignored
- Can throw an exception (any type) with `throw`
  - But usually they should inherit from `std::exception`
- You can catch an exception within a `try` block with `catch`.
- If you don't catch the exception, it is passed to the caller and then the caller's caller, etc. until it is caught
- Simple example at <https://godbolt.org/z/gJ97bS>

# Why do exceptions break explicit resource management?



- In the following example, the code to release resources is never called if g() throws an exception!
- ```
void f() {  
    code_to_create_resources;  
    g(); // May throw an exception  
    code_to_release_resources;  
}
```



We need a better solution

- To solve this, we will need to understand the C++ object lifecycle
- The payoff will be that C++ programmers generally spend very little time on memory management



RAII

- “Resource Acquisition is Initialization”
- One of the most important idioms in C++
- RAII uses automatic duration objects to manage the lifetimes of dynamic duration resources
- RAII just means that we ensure that an automatic object’s destructor performs any needed cleanup



RAII example

- Remember, an object's destructor is always called at the end of its life
- So the following code will call x's destructor no matter how we leave f
- Since unique_ptr's destructor destroys the object it is managing and releases its memory, the following code will clean up the X that was created
- ```
void f() {
 auto x = make_unique<X>();
 g(); // No resource leak even if g throws
}
```

# What if we didn't want the X object deleted



- Sometimes you want an object to outlive the RAII object that manages it
- In that case, move ownership to another RAII object
- ```
unique_ptr<X> f() {
    auto x = make_unique<X>();
    g();
    return move(x); // Transfer ownership to caller
}
```
- ```
void h() {
 unique_ptr<X> h_x = f();
 /* Use the X constructed in f(). It will
 be destroyed by h_x's destructor when
 h finishes */
```
- This is why nearly all RAII classes are movable



# RAlI Classes

- Just like the classes we know about (or will see today), `unique_ptr`, `shared_ptr`, `jthread`, `lock_guard`, `scoped_lock`, `unique_lock`, and `shared_lock` all use their destructor to release the object they manage
- Let's take a look at what can go wrong if we don't use RAlI classes



# What if we didn't use RAII?

```
std::list<int> some_list;
std::shared_mutex some_mutex;

void
add_to_list(int new_value)
{
 some_mutex.lock();
 some_list.push_back(new_value);
 some_mutex.unlock();
}

bool
list_contains(int value_to_find)
{
 some_mutex.shared_lock();
 auto result =
 std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
 some_mutex.shared_unlock();
 return result;
}
```



# Why is this worse?

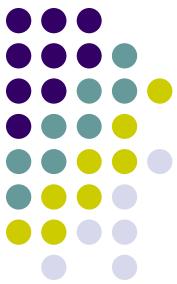
- Minor reason: It's more awkward
  - We can't just return the result of find. We have to store it in a local variable, then release the lock, then return the local
- Big reason: We could forget to unlock
  - It is easy to get manual resource management wrong
    - E.g., what if you added a return statement in an if in the middle of the function. It would be easy to forget to unlock
  - We've discussed this around unique\_ptr, but it applies to locks and other resources as well

# Huge resource management challenge: Exceptions



- Since an exception can disrupt the normal flow of control at almost any time, it may bypass our cleanup code
- For example, `vector::push_back` may throw a `bad_alloc` exception

# Let's see what goes wrong in the non-RAII code

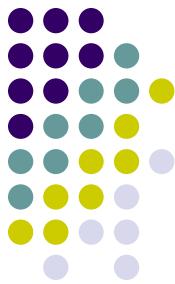


```
std::list<int> some_list;
std::shared_mutex some_mutex;

void
add_to_list(int new_value)
{
 some_mutex.lock();
 some_list.push_back(new_value); // May throw bad_alloc exception
 some_mutex.unlock(); // Oops! unlock() not called
}

bool
list_contains(int value_to_find)
{
 some_mutex.shared_lock();
 auto result =
 std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
 some_mutex.shared_unlock();
 return result;
}
```

# Using RAII to write exception-safe code



- ```
void f() {  
    auto a = make_unique<A>();  
    code_to_work_with_a  
}
```
- The A object is guaranteed to have its destructor called and memory cleaned up even if an exception is thrown
 - Because a doesn't exist after we leave f
 - So a's destructor will be called when we leave f (however we leave f)
 - a is a unique_ptr<A>, so its destructor cleans up the managed object
- As a bonus, the code is simpler and less error-prone because we no longer need to write the *code_to_cleanup*

Is our RAII example fixed?

Yes!



```
std::list<int> some_list;
std::shared_mutex some_mutex;

void
add_to_list(int new_value)
{
    std::unique_lock guard(some_mutex); // Will always be destroyed when we leave scope
    some_list.push_back(new_value);    // Even if an exception is thrown
}

bool
list_contains(int value_to_find)
{
    std::shared_lock guard(some_mutex);
    return
        std::find(some_list.begin(),some_list.end(),value_to_find) != some_list.end();
}
```

What if we want our object to outlive the automatic scope?



- It might be that we want our dynamic object to be longer-lived than the unique_ptr or unique_lock or ... that is managing it
- Just transfer ownership to another RAII object
 - `up2 = move(up);`
- This allows us to chain into new scopes while always maintaining an owner of the object

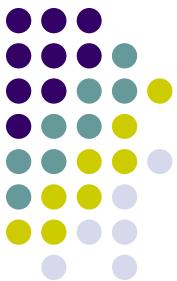


EXAMPLE: DISTRIBUTED COUNTER



Distributed Counter Example

- Let's explore concurrency best practices by looking at ways to implement a "distributed counter"
- This is a counter that can be incremented or read by any thread
- While this sounds like as simple of a problem as you can have in concurrency, we will see that it has a lot of complexity



Our first distributed counter

- The DistributedCounter1.h simply has a shared count protected by a mutex for incrementing and reading



mutable

- We made `mtx` a `shared_mutex` field of `DistributedCounter` to keep different `DistributedCounter` objects from interfering with each other
- You may have noticed the new word `mutable` in the declaration
 - `shared_mutex mutable mtx;`

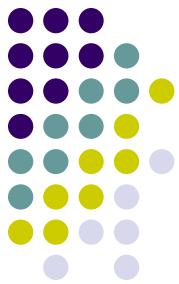


Logical constness

- We would like `DistributedCounter::get()` to be `const` because it seems illogical that reading the current value modifies the object
- However, getting the reader lock will modify the state of the `mtx` member of the counter object
- Since we don't want consider changes to `mtx` to be changes to the state modeled by our counter, we label it `mutable`, which means that it can be modified even in `const` contexts

DistributedCounter1

performance

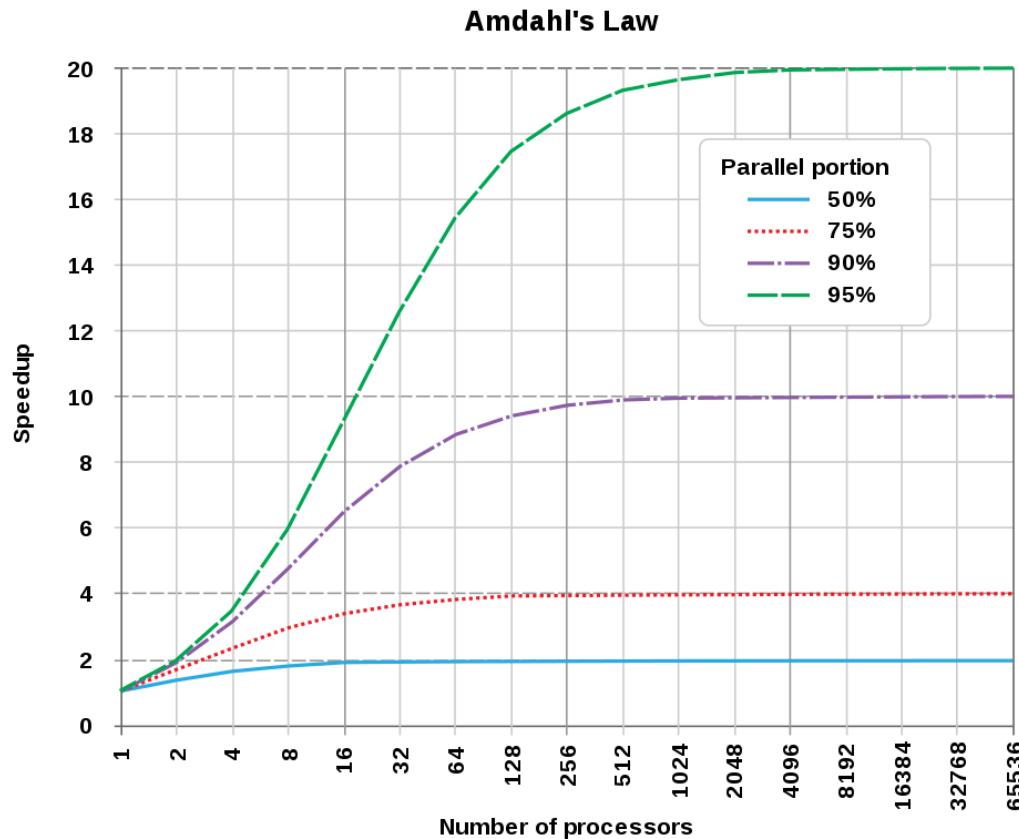


- This is simple, but very slow for two reasons
 - Amdahl's Law
 - Cache management

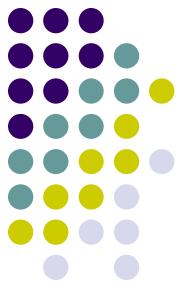


Amdahl's Law

The more sequential tasks in a program, the less it benefits from parallelism
Let's look at https://en.wikipedia.org/wiki/Amdahl%27s_law



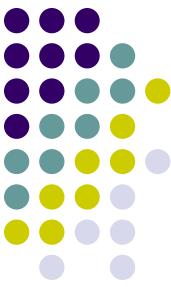
DistributedCounter1 and Amdahl's Law



- Because only one thread can increment the counter at a time, applications that increment the counter cannot take advantage of parallelism very well by Amdahl's Law



CACHE-CONSCIOUS PROGRAMMING

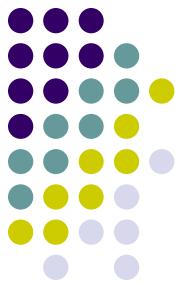


Cache effects

- Accessing main memory can take a processor hundreds of cycles
- Therefore, processors use high-speed caches to maintain local copies of data
 - See https://www.researchgate.net/profile/Philip_Machanick/publication/280154268/figure/fig1/AS:614268508590082@1523464433037/A-typical-mass-market-multicore-design-with-shared-third-level-L3-cache-Levels-1-L1.png
 - If another processor needs to read/write that memory, it needs to force other processors to flush or invalidate any cached copies of the memory
 - See http://en.wikipedia.org/wiki/Cache_coherency

DistributedCounter1

and caching



- Since every thread reads and writes the counter on every increment, caching is useless because each processor's cached value might be out-of-date
- When it modifies the counter, the computer's cache coherence protocol tells all other cores to discard their cached counter value



Cache lines and false sharing

- DistributedCounter2 attempts to fix this by hashing each thread to a different counter
 - This gives some speedup
 - But it is inconsistent and not as fast as we might hope
- Let's understand what went wrong
- When data is moved from main memory to cache, enough data is always moved to fill a “cache line.”
 - The size of a cache line varies by processor and needs to be looked up in the processor datasheet. A typical size would be 32 bytes, but it varies greatly.
- As a result, if two processors are modifying data within 32 bytes, they are constantly forcing each other to invalidate their cache (“false sharing”)



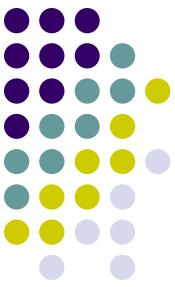
False sharing example

- Look at `DistributedCounter2` in `Canvas`
- Even though it uses multiple subcounters, since they are stored in an array, many of them end up in the same cache line, which means updating a counter on one thread means that all of the other threads will have to reload their counters from main memory since they are in the same cache line
 - This is very slow. Maybe 100x slower than accessing cache memory
- This kind of coupling of seemingly independent variables because they reside on the same cache line is known as **false sharing**

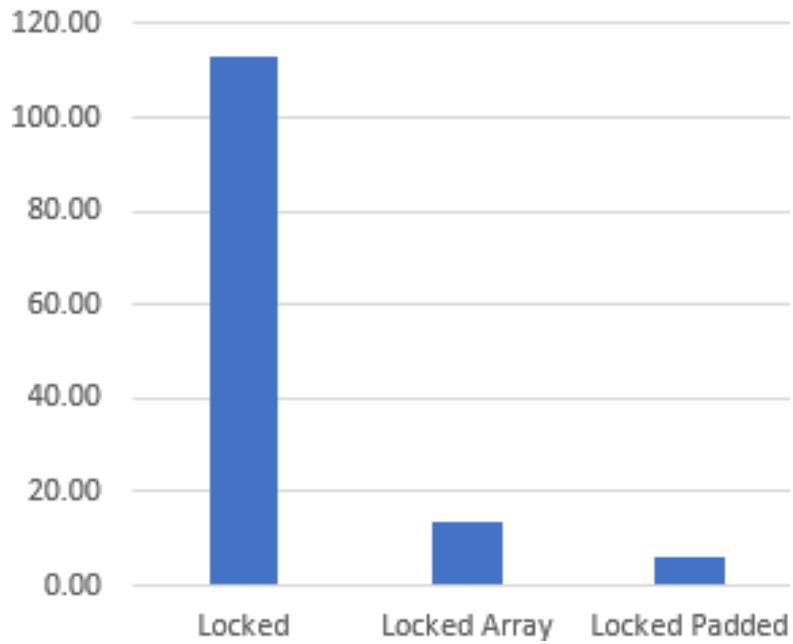


Eliminating the false sharing

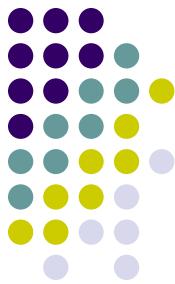
- In DistributedCounter3, we add some padding to the counters so they all are far enough apart to fall in distinct cache lines
- On my dual socket workstation with 10 threads per CPU, the program gets ~15 times faster!



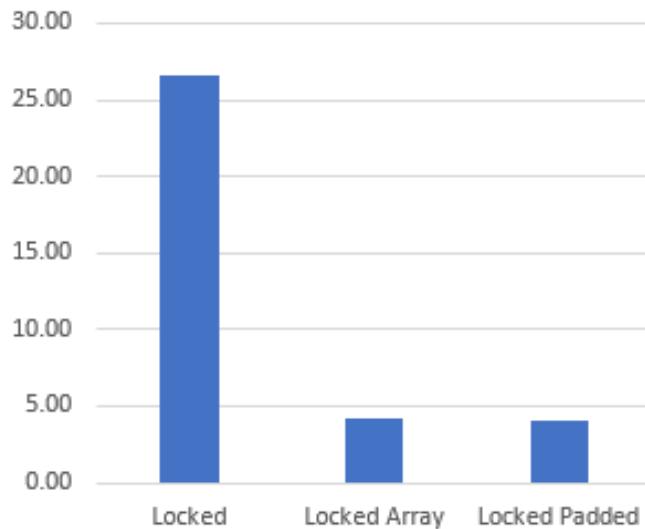
Let's recap



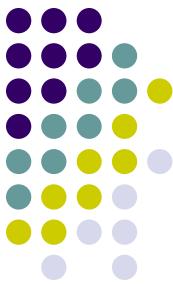
Are we done? Not so fast



- Here's the numbers for the same tests run on my laptop
 - I adjusted the total counts and numbers of threads but the code is the same



Can different machines give different behaviors?



- Yes!
- On my laptop, the padding has no effect!
- Since my laptop only has one CPU socket, the impact of false sharing is much less than on my production server which has two CPU sockets that can only communicate via the motherboard and main memory
- If I had developed my code only on my development laptop, I would have unknowingly paid a 100% performance penalty on my production server!
- This is why you have to follow best practices like putting independent objects on independent cache lines



Direct-Mapped Caches

- Often a single memory location can only be mapped to one or two possible cache lines
 - See <https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec15.pdf>
- Not understanding direct-mapped caches can have dire consequences
 - See next slide

Important real-world example of direct-mapped caches going haywire



- The popular postscript rendering program ghostscript was originally written by Peter Deutsch, who wrote a custom memory manager. It is certainly true that malloc()/free() performance is critical in postscript and Peter Deutsch was a memory management expert, having coauthored the first high-performance Smalltalk implementation.
- Peter Deutsch used a custom allocator that stored free pages stored in a linked list
- Tests 10 years later showed that ghostscript's memory manager was actually slowing postscript processing down by 30%



What went wrong?

- The custom allocator maintained a pool of free-pages in a linked list, with the first word of each free page as a pointer to the next free page. As this code was developed on a machine without a direct-mapped cache, it ran fine. However, on machines with direct-mapped caches, all of the freelist pointers mapped to the same cache line causing a cache miss on each step of walking through the freelist. Ghostscript was spending about a third of its time in cache misses from walking through the page freelist.
- Note: You don't need to understand this as long as you understand the moral

Cache-conscious programming

(Adapted from Herlihy&Shavit p. 477)



- Objects or fields that are accessed independently should be aligned and padded so they end up on different cache lines.
- Keep read-only data separate from data that is modified frequently.
- When possible, split an object into thread-local pieces. For example, a counter used for statistics could be split into an array of counters, one per thread, each one residing on a different cache line. While a shared counter would cause invalidation traffic, the split counter allows each thread to update its own replica without causing cache coherence traffic.
- If a lock protects data that is frequently modified, then keep the lock and the data on distinct cache lines, so that threads trying to acquire the lock do not interfere with the lock-holder's access to the data.
- If a lock protects data that is frequently uncontended, then try to keep the lock and the data on the same cache lines, so that acquiring the lock will also load some of the data into the cache.
- If a class or struct contains a large chunk of data whose size is divisible by a high power of two, consider separating it out of the class and holding it with an `unique_ptr` to avoid the Ghostscript problem from the previous slide
- Use a profiling tool like VTune to identify where your cache bottlenecks are

Portable cache-conscious programming



- This week's rules for cache-conscious programming, had a lot of phrases like “put on same/different cache line”
- How do we do that in C++?
- We could make an educated guess, like we did with the padded counter
 - However, such “magic numbers” are always brittle
- C++17 introduces a portable approach
 - If you keep objects further than `std::hardware_destructive_interference_size`, they “should” (i.e., implementation-defined) end up on different cache lines
 - If you keep objects closer than `std::hardware_constructive_interference_size`, they “should” end up on the same cache line



HW 5-1

- Try all the distributed counters on Canvas on your computer
- How do their performance compare?
- What do you conclude?
- Do you think your results would be the same on other computers? Why or why not?
 - Hint: Think about the considerations discussed in this lecture or add your own



HW 5-2

- Just like one can leak memory or forget to release locks, it is also easy to forget to join a thread
- This suggests an RAII class for threads would be useful
- C++20 added an RAII version of thread called `jthread`
- Change our `HelloThreads` and `DistributedCounterTest1.cpp` to use `jthread` instead of `thread`
- Do you like this version better?



HW 5-3

- Write a thread-safe stack using locks
 - The only required operations are push and pop
- E.g., multiple threads can concurrently do things like the following without corrupting the stack
 - `mpcs51044::stack<int> s;`
`s.push(7);`
`s.push(5);`
`cout << s.pop();`
- For extra credit, add additional useful functionality (e.g., initializer list constructor, etc.)



HW 5-4

- Create a Counter class template wrapper that can be used to count calls to any function. E.g..
 - ```
int f(double d)
{ return static_cast<int>(2*d); }
Counter cf(f);
```

```
int main()
{
 cf(1.3); cf(2.4);
 cout << format("f was called {} times\n", cf.count());
}
```
- Hint: Perfect forwarding and CTAD are your friends



# HW 5-5: Extra Credit

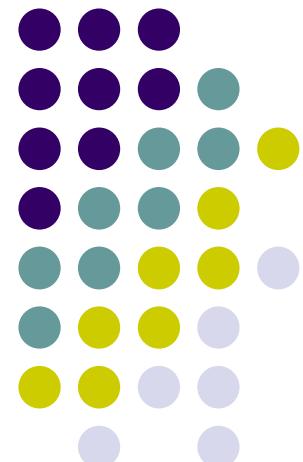
- While 5-4 allows us to count function calls, it isn't very transparent
- Switching between regular calls to `f` and counted calls to `f` would require us to change all the calls in `main()` between `f` and `cf`
- Can you rework the code so that you can change back and forth between regular calls and counted calls without needing to change `main()` each time you switch?
- Hint: “Every problem in programming can be solved with another layer of indirection”
- You can submit a single solution for both 5-4 and 5-5

# C++

## February 9, 2023

Mike Spertus

[spertus@uchicago.edu](mailto:spertus@uchicago.edu)



MASTERS PROGRAM IN  
COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO



# Final Exam/Final Project

- You can take either the final exam or do a project
  - 90%+ do the final exam
  - If you do both, you get the better score (but hardly anyone does)
- Final Exam
  - As mentioned earlier, the final exam consists of some questions similar to the homework and then a large code review
  - You will not use a compiler (think of it as a pencil and paper exam where you can use a code editor)
  - Exam will be in person but online
    - I will be sure to have sufficient electrical outlets
- Final Project
  - Any topic OK, but you need to show you learned much (not all) of the material in the class
  - E.g., if there are no threads, unique\_ptr, or templates, you won't do well
  - Sync with me in advanced



# LOW-LEVEL AND SYSTEMS PROGRAMMING



# Pointers

- Pointers are a low-level way to refer to a (typed) location in physical memory
- For the most part, one should use references and higher-level lightweight abstractions like `unique_ptr`
  - Why? (You'll see why in your homework)



# When do we need pointers

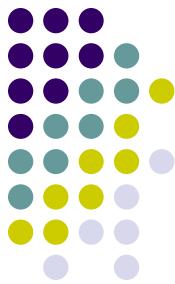
- There are some cases not covered by our lightweight abstractions
  - A reference always refers to the same object but doesn't own it and cannot be "unbound"
  - A unique\_ptr can change what object it manages, and can be unset (not managing any object), but always owns any object it manages
  - What if we want to be unsettable/rebindable but not own the object
    - Oops!
    - But see observer\_ptr for a possible future solution
    - [https://en.cppreference.com/w/cpp/experimental/observer\\_ptr](https://en.cppreference.com/w/cpp/experimental/observer_ptr)

# When do we need pointers (part 2)



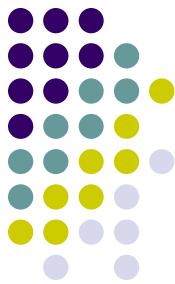
- C++ forces pointers on you in a number of cases
  - this is a pointer
    - Although it's generally better to use the reference `*this`
- `main(int argc, char **argv)` uses “pointers to pointers” to pass command line arguments
  - Yuck!

# When do we need pointers (part 3)



- Low-level programming
  - C++ is a systems programming language
  - You may need to access memory directly
  - For example, a temperature sensor external to the program may write the current temperature into a particular memory location, and your program needs to read the value at that known memory location

# When do we need pointers (part 4)



- Advanced memory management
- Nearly all memory management can be handled with `unique_ptr` and `shared_ptr`
- But occasionally there are situations they don't cover
- For example, managing the lifetime of nodes in a doubly linked list has shared ownership but using `shared_ptr` would be too inefficient



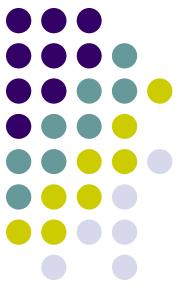
# Pointers

- Pointers to a type contain the address of an object of the given type.  
`A *ap = new A();`
- What does this mean?
- ap contains the address of an object of type A in memory
- new A() constructs an object of type A in memory and returns its address
- new A() has the same relation to A \* as `make_unique<A>()` has to `unique_ptr<A>`
- `A a;`  
`ap = &a; // & is the addressof operator`



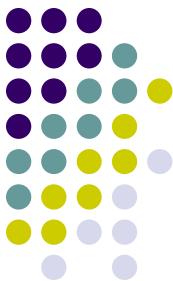
# Pointers

- Dereference with \*  
`A a = *ap;`
- `->` is an abbreviation for `(*_)`.  
`ap->foo(); // Same as (*ap).foo()`
- All of these should be familiar from `unique_ptr` (which intentionally provides a pointer-like interface)
- **Warning:** Unlike `unique_ptr`, raw pointers do not have destructors and do not manage the lifetime of the object they point to. They simply store its address



# nullptr

- If a pointer is not pointing to any object, you should make sure it is nullptr
  - ap = nullptr; // don't point at anything
  - if(ap) { ap->foo(); } // foo won't be called
- The type of nullptr is nullptr\_t, which implicitly converts to any pointer type



# Accept no substitutes

- Old C/C++ code often uses 0 or the NULL macro to represent an empty pointer.
- Don't do this, as it breaks overloading
- ```
int f(char *);  
int f(int);  
auto x = f(0); // Which f was meant?
```
- Solution: Don't every use 0 for nullptr
 - ```
auto x = f(0); // Calls f(int)
auto x = f(nullptr); // Calls f(char *)
```



# Pointer arithmetic

- Arithmetic on pointers

```
// Arrays decay to a pointer to the 0th element
A *aap = new A[10];
(*aap)[5] == *(aap + 5) // 5th elt of array
```

- It doesn't add 5 to the address, but adds enough to get to the fifth element (starting from 0), taking into account the size of the object
  - Note that you should avoid such low-level arrays whenever possible in favor of lightweight abstractions like `std::array` and `std::vector`



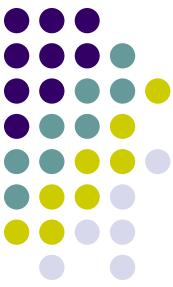
# C Strings

- C-style string literals give you pointers
- "foo" is actually a literal of type `char const[4]`
- Sort of works
  - `auto cp = "foo"; // c is a char const *`  
`cout << cp[1]; // prints o`
- But obviously not nearly as powerful or typesafe  
a C++ strings, which have many methods,  
addition, etc.
- Using C++-string literals is a good habit
  - `auto s = "foo"s; // s is a std::string`



# string\_view

- There are a lot of ways to express text in C++
- A std::string
- A C string (i.e., char const \*)
- A pointer and a length
- While it is possible to create a std::string from any of these, it can be clumsy and inefficient because all the character data will be copied
- What is a lightweight abstraction that can work with any version of text?



# string\_view

- ```
void f(string const &);  
f(string(foo)); // OK  
f("foo"s); // OK. Same with user-defined literal  
f("foo"); // Has to copy text into string  
f({"foobar", 3}); // Has to copy text into string
```
- Fine for short strings like this, but what if they were long?



string_view

- A `string_view` wraps a `char const *` and a `length`
- ```
void f(string_view);
f(string(foo)); // OK. Views buffer
f("foo"s); // OK.
f("foo"); // OK. No more copying text
f({"foobar", 3}); // OK. No more copying text
```
- The hope is that functions that now take a `string const &` or a `const char *` will take a `string_view` instead

# string\_view to string conversions

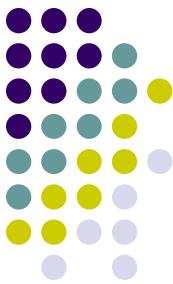


- While it is easy to go from a string to a string\_view, it can be hard to go the other way around
- ```
string s("foo");
string_view sv(s); // OK
string s2(sv); // OK: string_view to string
string s3 = sv; // Ill-formed!
```
- What went wrong? The string constructor used for s3 is explicit, so it doesn't generate an implicit type conversion



How bad is that?

- At first glance, that doesn't sound so bad
- If you need to initialize a string from a `string_view`, just use direct initialization like `s2` rather than copy initialization like `s3`
- Unfortunately, function arguments in C++ use copy initialization
- ```
void f(string const &s);
string_view sv("foo");
f(sv); // Ill-formed! Copy initialization
```
- Solution: `f` should take a `string_view`



# Raw string literals

- When you have a long string, is sometimes painful to constantly escape internal quotations and backslashes
- Furthermore, a quoted string cannot extend across multiple lines (unless you put \n in)
  - The point was to catch if you forgot to close your quotation marks



# Raw string literals

- C++ raw string literals
- General form is R"delim(*text*)delim"
- What is *delim*?
- Anything you want. Use it to avoid collisions
- ```
string s = R"foo(Hello  
(Raw String) Literals)foo";
```
- You can skip the delimiter if you don't need it

```
string code =  
R"(#include<iostream>  
using namespace std;
```

```
int main()  
{  
    cout << "Hello, world\n";  
}");
```



Pointers to functions

- The basic idea is usually that you describe a type by how it is used
 - `int *ip; // Means *ip is an int`
 - `int (*fp)(int, int); // *fp can be called with 2 ints`
- Let's show fp in action
 - ```
int f(int i,int j) { ... }
fp = &f;
fp(2, 3);
// The following line only works without captures
fp = [](int i, int j) { return i + j; }
```



# Function pointers: motivation

- Sometimes we don't know what function we want to call until runtime

- ```
double mean(vector<double> const &) { ... }
double median(vector<double> const &) { ... }
cout << "Should I use means or medians ";
string answer;
cin >> answer;
double (*averager)(vector<double> const &)
= (answer == "mean" ? mean : median);
cout << "The average home price is ";
cout << averager(getHomePrices()) << endl;
```



Pointers to members

- ```
struct A {
 int i;
 int j;
 void foo(double);
 void bar(double);
};
```
- We would like to be able to point to a particular member of A
  - Not an address because we haven't specified an A object
  - More like an offset into A objects
- ```
int A::*aip = &A::i;  
void (A::*afp)(double) = &A::foo;  
A *ap = new A;  
A a;  
ap->*aip = 3; // Set ap->i to 3  
(a.*afp)(3.141592); // Calls a.foo(3.141592)
```



Pointer to member functions

- Consider

```
vector<Animal *> zoo;

zoo.push_back(new Elephant);
zoo.push_back(new Zebra);
zoo.push_back(new Bear);
cout << "Feeding time (f) or Bedtime (b)?"
char c;
cin >> c;
void (Animal::*ap)()
    = c == 'f' ? &Animal::eat : &Animal::sleep;

for(auto animal : zoo) {
    animal->*ap();
}
```

Using with standard smart pointers



- Unfortunately, `unique_ptr` and `shared_ptr` don't overload `operator->*`, so if we want to make the previous example delete objects when the zoo closes (or there is an exception when constructing an animal), we should modify it as shown below

```
vector<unique_ptr<Animal>> zoo;
zoo.emplace_back(new Elephant);
zoo.emplace_back(new Zebra);
zoo.emplace_back(new Bear);
cout << "Feeding time (f) or Bedtime (b) ?"
char c;
cin >> c;
void (Animal::*ap)()
= c == 'f' ? &Animal::eat : &Animal::sleep;

for (auto it = zoo.begin(); it != zoo.end(); it++) {
    ((*it).*ap)();
}
```



References

- Like pointers but different
 - Allow one object to be shared among different variables
 - Can only be set on creation and never changed
 - Reference members must be initialized in initializer lists

```
struct A {  
    A(int &i) : j(i) {}  
    A(int &i) { j = i; } // Ill-formed!  
    int &j;  
};
```
 - Cannot be null

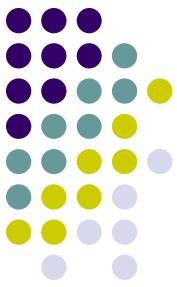
Not all callables can be assigned to a function pointer



- Can only assign a lambda to a function pointer if it does not have a capture list
 - See homework

- Can't assign a functor to a function pointer

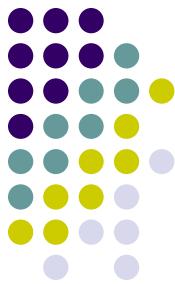
```
• struct WeightedMean {  
    WeightedMean(vector<double> const &weights)  
        : weights(weights) {}  
    double operator() (vector<double> const &data) {  
        return  
            inner_product(data.begin(), data.end(),  
                          weights.begin(), 0.0)  
            / accumulate(weights.begin(), weights.end(), 0.0);  
    }  
    vector<double> weights;  
};  
WeightedMean wm({1.2, 3.4});  
wm(getHomePrices()); // Fine  
double (*averager) (vector<double> const &)  
= WeightedMean({1.5, 3.6, 4.2}); // Error!  
cout << "The average home price is ";  
cout << averager(getHomePrices()) << endl;
```



std::function

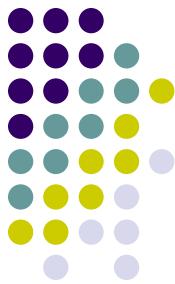
- We have just discussed function pointers, but in C++, functions aren't the only thing that can be called
 - Call a function
 - Call a lambda
 - Call a functor
 - Call a member function

std::function can hold anything callable



- ```
struct WeightedMean {
 WeightedMean(vector<double> const &weights)
 : weights(weights) {}
 double operator()(vector<double> const &data) {
 return
 inner_product(data.begin(), data.end(),
 weights.begin(), 0.0)
 / accumulate(weights.begin(), weights.end(), 0.0);
 }
 vector<double> weights;
};
function<double(vector<double> const &) > averager
= WeightedMean({1.5, 3.6, 4.2}); // OK
cout << "The average home price is "
cout << averager(getHomePrices()) << endl;
```

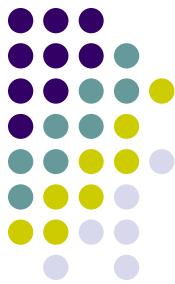
# You can even put a member pointer in a std::function



- It acts like a function whose first argument is the “this” pointer (or even a reference).

- ```
struct A {  
    int i;  
};  
function<int(A *)> fp = &A::i;  
A a;  
fp(&a);
```

Often you can choose between a std::function and a template



- In the below code, `tmpl_apply` and `fn_apply` can be used similarly

```
template<typename Callable>
double
tmpl_apply(Callable c, vector<double> const &data)
{
    return c(data);
}

double
fn_apply(function<double(vector<double> const &)> c, vector<double> const
&data)
{
    return c(data);
}
void f()
{
    tmpl_apply(mean, {1.7, 2.3}); // OK
    fn_apply(mean, {1.7, 2.3}); // OK
    tmpl_apply(WeightedMean({1.2, 3.4}), {1.7, 2.3}); // OK
    fn_apply(WeightedMean({1.2, 3.4}), {1.7, 2.3}); // OK
}
```

- See HW



How does unique_ptr work?

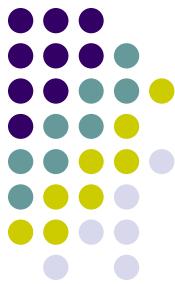
- Suppose we say
 - `auto ap = make_unique<A>(1, 2);`
- How does the object get created under the hood?
- First, `make_unique`'s calls
 - `new A(1,2)`
- The compiler's first step in this "new expression" is to allocate memory by calling
 - `operator new(sizeof(A))`
- Then it constructs an `A(1, 2)` object in the memory that was allocated



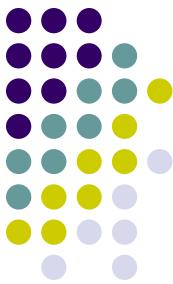
operator new

- 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's
 - 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 your entire program just overload the regular operator `new` to call that



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 = g(make_unique<T>());
auto p2 = g(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
```



How to get rid of an object

- When you are done with an object, it needs to be destroyed and its memory released
- `delete` takes a raw pointer to an object, calls its destructor, and then releases its memory
 - Using operator `delete()`
- `Animal *ap = new Elephant();
ap->eat();
delete ap; // Note: Missed if eat() throws an exception`



ATOMICS



Avoiding locks

- Whenever you have a lock, the critical section of code that it protects cannot be parallelized
- Fortunately, C++ has another mechanism, *Atomics*, that can let you write multithreaded programs without locks



C++ atomics

- Sometimes you just want a variable that you can read and update from multiple threads
- Using locks seems a little too complicated for that
- Especially because x86 has a built-in atomic increment
- Fortunately, C++ has a library of atomic types that can be shared between threads



An atomic counter

- You can read an atomic with its `load()` method, write it with its `store()` method and (usually) increment or decrement it with `++` or `--`
- See `DistributedMethodCounter4` for an updated version our Distributed Counter that uses atomics



CASE STUDY ON THE RISKS AND REWARDS OF LOW- LEVEL PROGRAMMING

Background: How to quickly allocate objects of a fixed size?

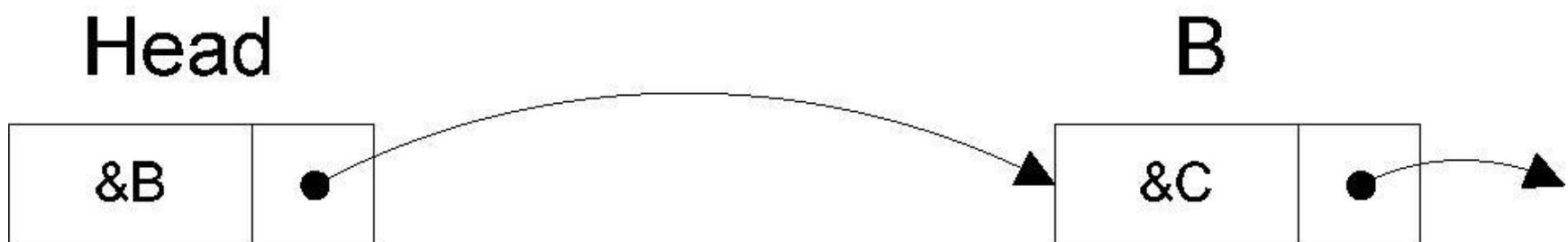
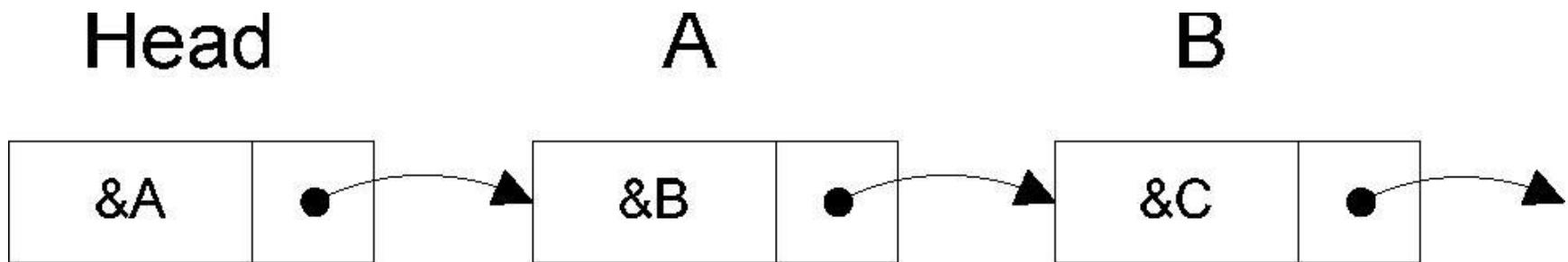


- Say we're allocating 32-byte objects from 4096-byte pages
- Divide each page in our memory pool into 128 objects in a linked list
- Now, allocate and deallocate 32-byte objects from the list by pushing and popping
 - Fewer than a dozen instructions vs hundreds in a conventional allocator
 - Make sure you lock for thread-safety
- You have implemented such a lock-based stack as an exercise



Allocating an object

- Pop the first object off the list

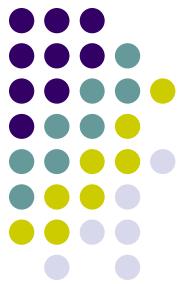


A True Story with a Twist—The Bad Beginning

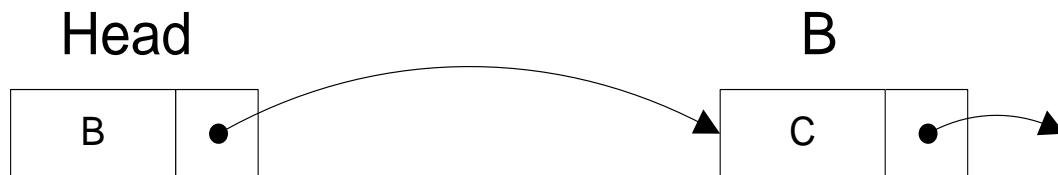
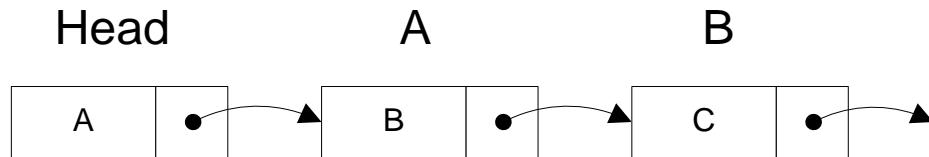


- A programmer released an application using a linked list allocator like in the previous slide
 - It appeared to speed up his program considerably
- His customers reported that the application became slow as the number of threads increased into the hundreds
- Even though the lock only protects a few instructions, if a thread holding the lock loses its quantum, the list is unavailable until that thread gets another timeslice (perhaps hundreds of quanta later)
- Not acceptable

Can we make a thread-safe list without locks?



- To Remove an element

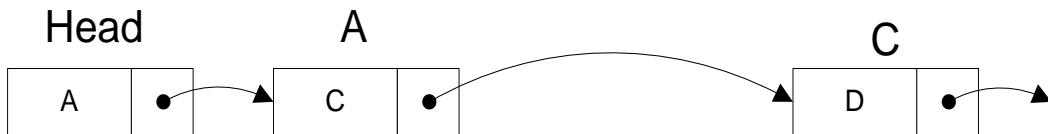
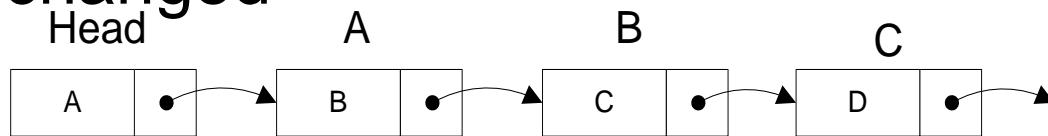


- We need a lock because we need to both return A and update the head to point to B (i.e., A's link) atomically
- Or do we?
- C++ has an atomic `compare_and_exchange_weak` primitive that does a swap, but only if the target location has the value that we expect
 - Then our update would fail if someone messed with the list in the critical section
 - If so, just loop back and try again

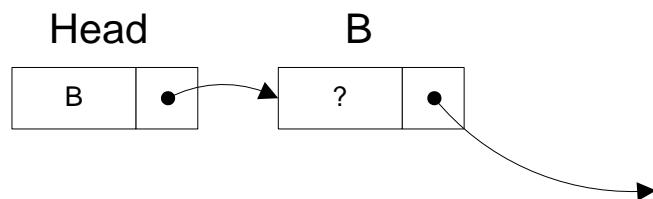


Oops! Doesn't quite work

- Some other thread could do two pops and one push during the critical section, leaving the head unchanged



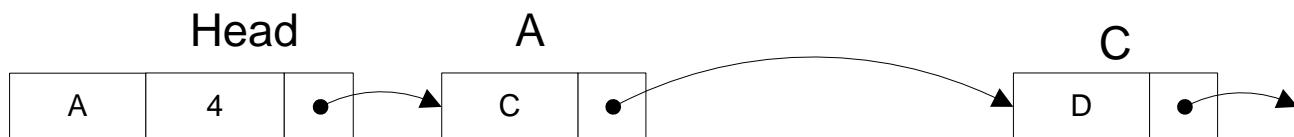
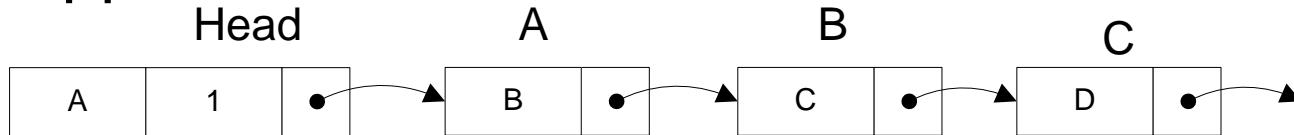
- After the `compare_and_exchange_weak`, B is erroneously back on the list

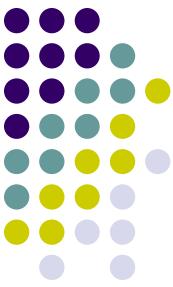




We can fix this

- Add a “list operation counter” to the head
- Update with 64-bit compare and exchange (on a 32-bit program), which C++ conveniently provides (and maps onto a single x86 instruction provided for just this reason)
- Now the compare and swap fails if intervening list ops happened

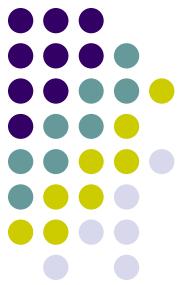




What's the point?

- This is much better
- No need for memory barrier
- Only one atomic operation instead of two
- If thread loses its quantum while doing the list operation, other threads are free to manipulate the list
 - This is the big one
- Works on x86-32, x86-64, and Sparc

How is this implemented in C++?



- See lockFreeStack.h in chalk
- Let's look at it now



What about PPC and Itanium?

- Even better, PPC and Itanium have Linked Load and Store Conditional (LLSC)
- lwarx instruction loads from a memory address and “reserves” that address
- stwcx instruction only does a store if no intervening writes have been made to that address since the reservation
- Exactly what we want



What about push?

- The same techniques work for pushing onto the list
 - Exercise to see if you understand
- Not just restricted to lists
 - Many other lock-free data structures are known
 - See the references

A True Story with a Twist—A Happy Ending?



- The programmer switched to using Compare and Exchange-based atomic lists on Sparc
- The customers were happy with the performance
- But wait...



No happy ending?

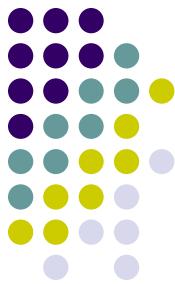
- The customers started to experience extremely intermittent list corruption
- Virtually impossible to debug
 - He ran 100 threads doing only list operations for hours between failures
- The problem was that Solaris interrupt handlers only saved the bottom 32-bits of some registers
 - Timer interrupts in the critical section corrupted the compare and exchange
 - Fix: Restrict list pointers to specific registers
- Moral: The first rule of optimization is “Don’t!”
 - These techniques are powerful but only used where justified



But wait, there's more

- Later, the program started being used on massively SMP systems, and it started to exhibit performance problems
 - The Compare and Exchange locked the bus to be thread-safe but that is expensive as the number of processors went up (this results in a surprising implementation of the Windows Interlocked exchange primitive).
- Since they no longer needed many more threads than processors, they went back to a lock-based list

So should you customize your memory allocation?



- Do you really want to pollute your class with deep assumptions about the HW and OS?
- Do you want to update it everytime there is a new OS rev?
 - Early version of this before threading inadvertently made classes thread unsafe
- The answer is almost always, “No,” but...

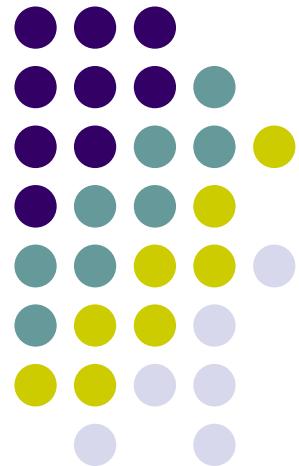
No way!

Except...



- My friend's product wouldn't have been usable without a custom memory manager
- He wouldn't have sold his company for a large sum of money without usable products
- Use it when necessary, but only if you can justify the costs of maintaining your code over every present and future OS/hardware revision
- This story illustrates the real power and danger of using C++
 - Know the difference between “use” and “abuse”

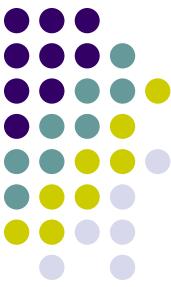
Homework





HW 6-1

- Use what you have learned about modern C++ from this course to improve the binary tree class in Canvas taken from
<http://www.cprogramming.com/tutorial/lesson18.html>
- In particular, think about whether the class' use of raw pointers is appropriate and fix if needed
- For full credit, think about whether it should have copy constructor and write them if appropriate
- Do you think your modifications are an improvement? Why or why not?



Homework 6-2

- This problem consists of a series of types. Write a program that defines variables of each type set to some meaningful value (You are highly encouraged to check with a compiler). If the type is callable, the program should call it. Googling “c++ declarators” may help. Each one you get is worth 2 points.
- Example problem 1: `int *`
 - One possible answer:
`int *ip = new int;`
 - Another possible answer
`int i = 5;`
`int *ip = &i;`
- Example problem 2: `int &`
 - One possible answer:
`int i = 5;`
`int &ir(i);`



HW 6-2 (cont)

- int *
- int &
- double
- A * (A is any appropriate class).
- char const *
- char const &
- long[7]
- int **
- int *&
- float &
- int (*) () (See <http://www.newty.de/fpt/index.html>)
- int (*&) ()
- char *(*)(char *, char *)



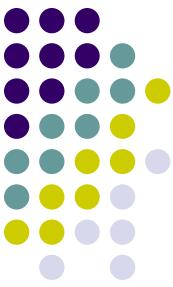
HW 6-2 (cont)

- int A::*
- int (A::*) (int *)
- int (A::**) (int *)
- int (A::*&) (int *)
- int (A::*) (double (*) (float &))
- void (*p[10]) (void (*)());



HW 6-3: Extra Credit

- Write a program that makes essential use of as many of the above types as possible to do something meaningful
 - The phrase “something meaningful” will be interpreted very liberally. This problem is meant to be an opportunity to get a feel for the different types
 - Feel free to make “cosmetic” adjustments to the types. E.g., using a `double (*) ()` instead of an `int (*) ()`
 - Some of the examples of using member pointers in the slides above may be helpful here...



HW 6-4

- Complete the implementation of lock free stacks by implement the push() method in LockFreeStack.h (on Canvas).



HW 6-5

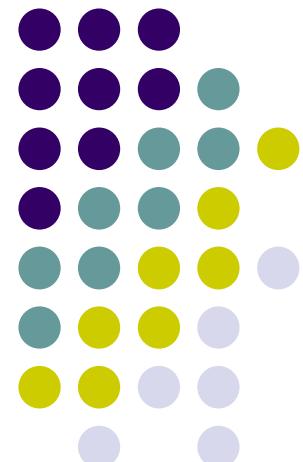
- Since this lecture is on low-level systems programming and memory, it is a good chance to remind ourselves that computer memory stores numbers in binary
- Learn to count in binary on your fingers
 - See http://en.wikipedia.org/wiki/Finger_binary
 - We'll test this in class
- How high can you count on both hands?
- Extra credit: Count to 31 in 15 seconds or less

C++

February 16, 2023

Mike Spertus

spertus@uchicago.edu



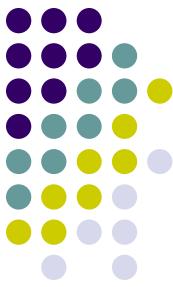
MASTERS PROGRAM IN
COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO



Sleeping a thread

- Sometimes you want to let a thread sleep for a period of time and then pick up its work
 - E.g., Aggregate statistics for a data structure every 30 seconds
- Traditional threading libraries like Win32 threads and pthreads have a sleep function. E.g., `sleep(30)`
- But how long will that sleep?
- 30 milliseconds? 30 seconds? 30 minutes?
- Well, you could look at the documentation, but in practice, you're not protected against errors, and people sometimes get it wrong



Can we do better in C++?

- The reason the traditional code is confusing is because it doesn't take advantage of the type system
- Using an integer to represent time intervals gives you no protection against accidentally misinterpreting it as some other concept represented by an integer
- As we've seen before, stronger types will allow us to write more robust code
- C++ is good at creating strong types, defining type conversions to avoid boilerplate, and using templates to get rid of overhead



std::chrono – clocks

- Clocks: Source of time information, providing
 - now – the current time
 - std::chrono::system_clock::now()
 - type – what type of value represents the time (unsigned, long, etc.)
 - std::chrono::system_clock::time_point
 - tick period – How often the clock ticks
 - If std::chrono::system_clock::period is std::ratio<1, 100>, it ticks 100 times per second
 - accuracy – Is the tick period consistent or is it an average
 - std::chrono::system_clock::is_steady



std::chrono – durations

- A duration is a length of time
- To get a value out, cast to the type that you want, and then use count
- `std::chrono::seconds(std::chrono::days(1)).count()` gives number of seconds in a day
- Unfortunately, the notation is cumbersome
- `sleep(std::chrono::milliseconds(30)); // yuck`



std::chrono – time points

- Points in time associated with a clock
- You can add or subtract a duration from a time point to get a new time point
- You can subtract one time point from another to get a duration provided the time points came from the same clock
- To get the current time_point, use `std::chrono::system_clock::now()`



User-defined literals

- C++ allows you to use suffixes to specify literals for built-in types
- `auto x = tuple(1, 2u, 3l); // int, unsigned, long`
- What about for classes?
- Let's consider why this is useful

User defined literals make it better



- C++ has user-defined literals that lets you use suffixes to build objects
 - To get 40 milliseconds, just say `40ms`. Likewise
 - `1y + 2d + 3h + 7min + 5s + 2ms + 5us + 6ns` means 1 year, 2 days, 3 hours, 7 minutes, 5.002005006 seconds
- Now it's easy to make clear that I want to sleep 30 milliseconds
- `sleep(30ms);`



Sleeping a thread in C++

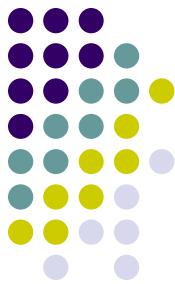
- Sleep for 30 seconds
 - `std::this_thread::sleep_for(30s);`
 - Clear we mean 30 seconds, not 30 ms or anything else
- C++20, also gives easy creation of time points.
 - `// Groundhog sleeps until noon on Groundhog Day
this_thread::sleep_until(sys_days{February/2/2019} + 12h);`
- This shows how strong types can wrap the underlying number representing a duration or point in time in a safe and clear fashion with no loss in performance



String literals

- What type is "hello, world"?
- It is a C string literal, so it is actually of type `char const *`
- Way too low level
- We can sort-of get away with it because there is a conversion from C strings to C++ strings, but things often go wrong
 - We'll see examples in a few slides
- To get a C++ string literal, just put an s on the end
 - "hello, world"s has type `string`

How do I define my own literal?



- What does a temperature of 7 mean?
 - Fahrenheit, Celsius, Kelvin?
- Let's write a temperature class
 - `struct Temp { double degrees_K; };`
 - `constexpr Temp operator""_K(double d)`
`{ return Temp{d}; }`
 - `constexpr Temp operator""_C(double d)`
`{ return Temp{d + 273.15}; }`
 - `constexpr Temp operator""_F(double d)`
`{ return Temp{(d - 32)*5/9 + 273.15}; }`
 - `static_assert(32_F == 0_C); // Units clear`
- Note: You can only write UDLs that begin with an underscore. Ones that don't are reserved for the standard library



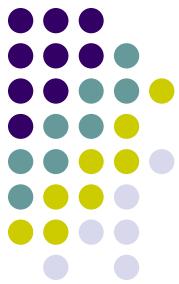
INTER-THREAD COMMUNICATION

Sometimes locks aren't what you want



- Suppose we are trying to implement a “producer/consumer” design pattern
 - Think of this as a supply chain
 - Some threads produce work items that are consumed by other threads
 - Incredibly common in multi-threaded programs
- Typically the producers put work onto a queue, and the consumers take them off
- Locks can allow thread-safe access to the queue
- But what happens if there is no work at the moment?
 - The consumer thread needs to go to sleep and wake up when there is work to do
 - Rather than a lock, you'd like to wait for an “event” stating that the queue has become non-empty

Producer-consumer implementation



- We will use a couple of new library features
 - `unique_lock`,
 - a richer version of `lock_guard`
 - `condition_variable`
 - “wakes up” waiting threads



Condition variables

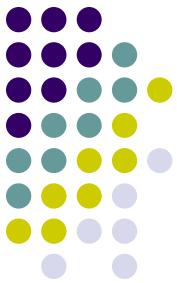
- The C++ version of an event
`condition_variable cv;`
- You can wait for a condition variable to signal
`mutex m;
boolean test();
unique_lock<mutex> lck(m);
cv.wait(lck, test);`
- If the test succeeds, the wait returns immediately, otherwise it unlocks m (that's why we used a `unique_lock` instead of `lock_guard`)
- Once the `condition_variable` signals the waiting thread (we'll see how in a moment)
 - The lock is reacquired
 - The test is rerun (if it fails, we wait again)
 - Protects against spurious wakeups
 - Once the test succeeds, the program continues



Signaling an event is simple

- `cv.notify_one();`
 - Wakes one waiter
 - No guarantees which one
- `cv.notify_all();`
 - Wakes all waiters

Producer-consumer from Williams' *C++ Concurrency in Action*



```
mutex m;
queue<data_chunk> data_queue;
condition_variable cond;

void data_preparation_thread()
{
    while(more_data_to_prepare()) {
        data_chunk const data=prepare_data();
        lock_guard lk(m);
        data_queue.push(data);
        cond.notify_one();
    }
}

void data_processing_thread()
{
    while(true) {
        data_chunk data;
        {
            unique_lock<mutex> lk(m);
            cond.wait(lk,[]{return !data_queue.empty();});
            data=move(data_queue.front());
            data_queue.pop();
        }
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

Async functions: Running functions in another thread

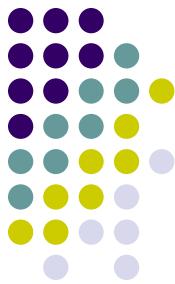


- It's nice that we can pass arguments to a thread (like we do to functions), but how can we get the thread to return a value back?
- Basically, we want to be able to use threads as “asynchronous functions”

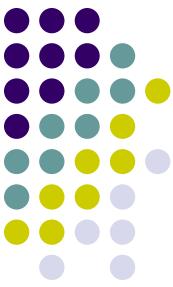
```
std::future<int> f = std::async(func_returning_int);
```

- C++11 defines a `std::future` class template that lets a thread return a value sometime in the future when its calculation is complete
- One way to create a future is with `std::async`
 - As soon as you create it, it starts running the function you passed it (usually) in a new thread
 - Call `get()` when you want to get the value produced by the function
 - `get()` will wait for the thread function to finish, then return the value

Introducing promises and futures



- Promises and futures let you pass values between threads
- Kind of like a producer/consumer but you are just passing a single item, so you don't need a queue
- A promise lets you pass a value from one thread to another
- Create a paired future and promise
 - `promise<int> p;`
`future<int> fi = p.get_future();`
- The producing thread stores the value in the promise
 - `p.set_value(7);`
- The consuming thread receives the value when it becomes available
 - `cout << "received " << fi.get() << endl;`



std::future example

- From [Multithreading in C++0x Part 8](#)

```
#include <future>
#include <iostream>

int calculate_the_answer_to_LtUaE();
void do_stuff();

int main()
{
    // Run calculation in a different thread
    std::future<int> the_answer
        = std::async(calculate_the_answer_to_LtUaE);
    do_stuff();
    std::cout << "The answer to life, the universe and everything is "
        << the_answer.get()
        << std::endl;
}
```

Can I check if the future has a value yet?



- Yep,
- `f.wait_for(std::chrono::seconds(0)) == std::future_status::ready;`
- Wow, that was ugly
- Can we do better?
- Yes. Use a literal!
- `f.wait_for(0s) == std::future_status::ready;`

What if the asynchronous function throws an exception?



- If the thread function in an `async` throws an exception instead of returning a value, then calling `get()` will throw the exception, just like the asynchronous function was a real function



Introducing packaged tasks

- Well, `async` is good if you just want to run a function on a new thread to get a return value,
 - but what if you want to asynchronously run a function on a specific existing thread, like a UI thread?
 - or if you need
- Well, that is what a `std::packaged_task` is for
- If I had said in one thread

```
packaged_task<int (double)> pt{f};  
future<int> fi = pt.get_future();
```
- Then I could run `pt` in any other thread

```
pt(3.6);
```
- And the value will become available in `fi`



Parallel accumulate

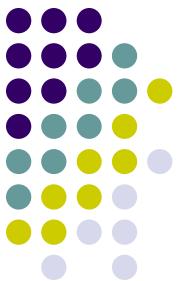
- It would be really nice to have an implementation of std::accumulate that breaks up its input into pieces, adds up each piece in parallel and then adds up the results from each of the pieces
- Let's do this with futures
- `async_accumulate_function.h`
 - Implementation due to Anthony Williams



Are futures an RAll class?

- In other words, does the destructor of a future join the thread if no one has called get() yet?

- ```
void f() {
 auto result = async(g);
 if(needResult())
 auto val = result.get();
 // If I didn't need the result,
 // will the return block until
 // result's destructor
 // joins g's thread?
 return;
}
```



# Answer: It Depends!

- If you get a future from a promise, it is “non-blocking”
  - Destroying the future does not wait for the thread to complete
- If you call `async`, it returns a “blocking future”
  - Its destructor will join the thread

# This causes some surprising behavior



- The following code runs in parallel

```
auto a = async(f);
auto b = async(g);
```

- The following code has no parallelism!

```
async (f) ;
async (g) ;
```

- The point is that since the future returned from `async` is not stored anywhere, it will be destroyed after line 1, so its destructor will wait for `f`'s thread to complete running before it advances to the next line and launches `g`!



# That's confusing!

- Yes. The committee agrees it is broken and will fix it in a future version of C++
  - No consensus yet on how
- In the meantime, you don't need to worry about any of this as long as you always call `get()` on a future sometime before destroying it (even if you don't really need to)
  - This covers most use cases



# C++23



# C++23 was finalized last week

Completed  
C++23

Issaquah, WA, USA  
(2023)



Photo – Frank Birbacher





# Large and small standards

- We generally don't release two "feels like a new language" standards in a row
- Big ones: C++98, C++11 and C++20
- Small ones: C++03, C++14, C++17, C++23
- But even the "small" ones have high-impact features
  - C++14: Generic lambdas: `[](auto x) { return x*x; }`
  - C++17: CTAD: `lock_guard lck(mutex);`  
Fold Expressions:  
`auto sum(auto const &...vs) { return (vs + ...); }`



# What about C++23

- For a broad writeup, see  
<https://mariusbancila.ro/blog/2022/12/23/the-cpp23-standard-break-down/>
- We're going to celebrate C++23 by digging into what I think will be a high impact feature
- std::expected
  - A new way to think about error handling
  - But first, let's dig a little deeper into exceptions

# Exceptions have a pervasive impact on your code



- As we've discussed, exceptions can cause clean-up code to be missed
  - ```
void old_fashioned_f() {
    X *x = new X("foo");
    x->g();
    delete x; // Oops! Skipped if g or h throws an exception
}
```
- We learned how to fix this with RAII
 - ```
void f() {
 auto x = make_unique<X>("foo");
 x->f(g()); // x cleaned up even if g or h throw an exception
}
```
- 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

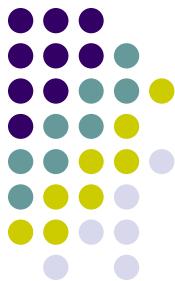
- Recall that an object's destructor is always called at the end of its lifetime
- That means destructors of automatic duration objects are called during exception processing even though normal code is bypassed
- This is what makes RAII work!
- But..

# 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

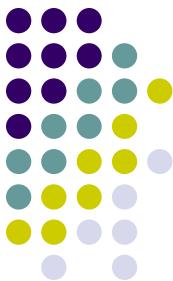


- To avoid this, don't throw exceptions in your destructor
- If your destructor calls something that may throw, be sure to catch before returning
- ```
struct X {
 ~X() {
 try {
 g(*this); // g may throw
 } catch(...) {} // Ignoring may be only option
 }
};
```

# 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 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!

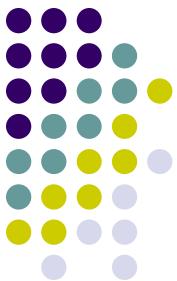


- Again, you would expect std::stack to be able to pop an object of a stack and return its value
  - `stack<A> stk;`  
...  
`A a(stk.pop()); // Illegal!`
- The problem with this would be if A's copy or move constructor threw an exception.
  - The top element could be lost forever
- Instead, `stack::pop`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

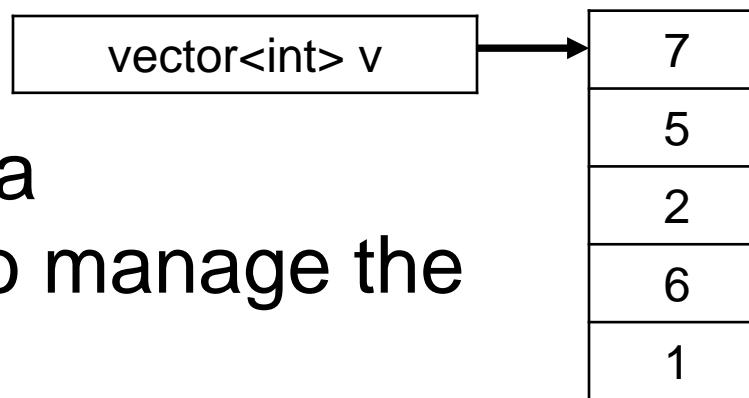


- As we just saw, `std::stack` has a convoluted interface imposed on it by exception safety
  - In particular, this interface is bad in multi-threaded programs (why?)
- Here's an even worse example where not knowing if an exception can be thrown forces inefficient algorithms to be used



# How do we grow a vector?

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

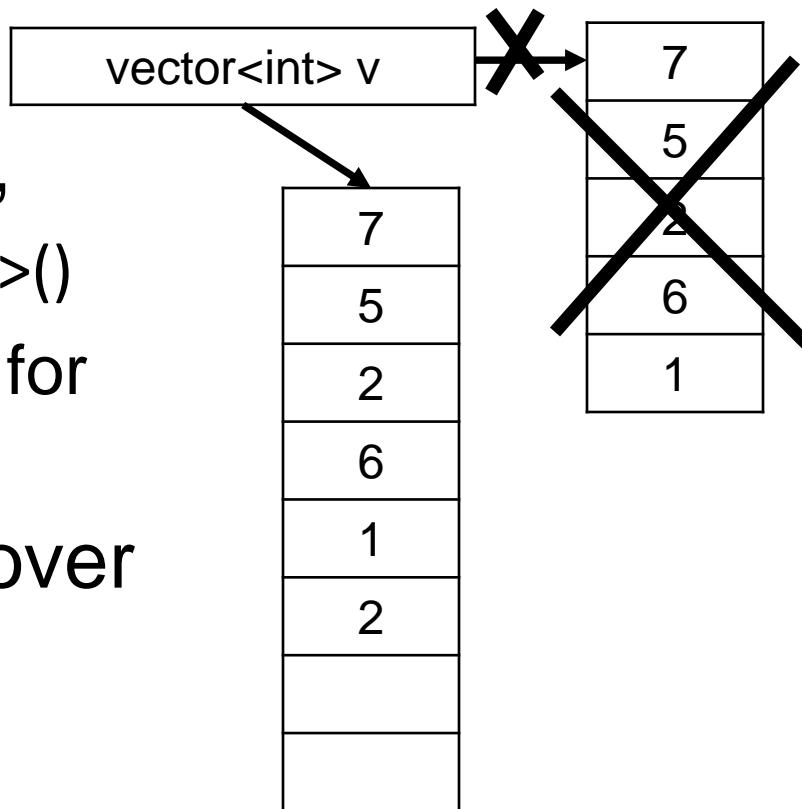


- (Logically) it uses a `unique_ptr<int[]>` to manage the array



# 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



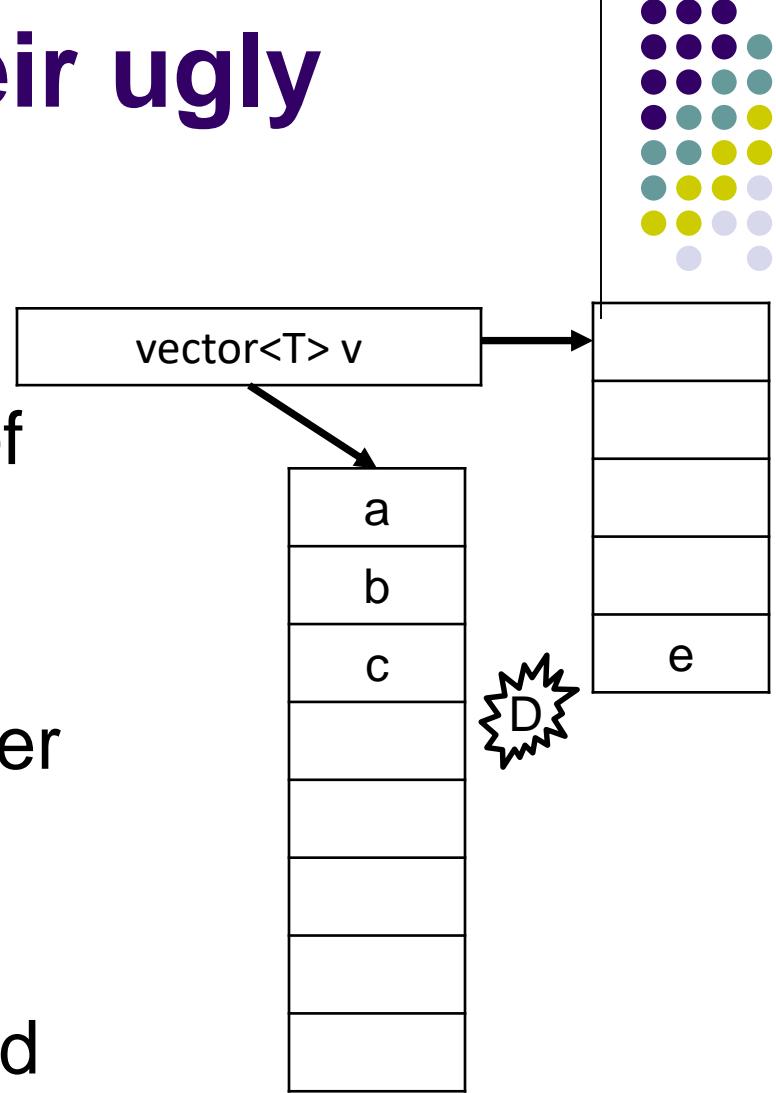


# Can we move?

- 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



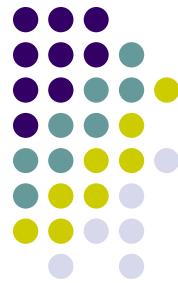
- In the previous slide, we would have been better off copying than moving
- But performance may be worse
- And something like `vector<unique_ptr<T>>` would simply be impossible
- This is, of course, unacceptable



# 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



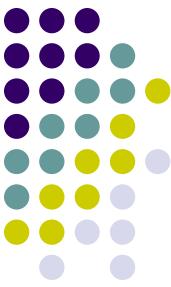
# Templates and noexcept

- 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!
- C++23 added a new way to handle errors: `std::expected`
- Let's learn about it!



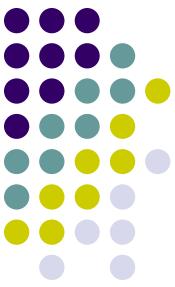
HW 7-1

- Use `std::packaged_task` to create your own implementation of `async`
 - You don't need to implement advanced `async` functionality that we haven't talked about in class, like launch policies.
 - In particular, don't worry about the earlier slides on whether the destructor should join the created thread. That is a highly technical point that we won't worry about here
 - For extra credit, learn about and implement these advanced features
- See the next slide for some background and hints
- Please don't base your solution off solutions on the web. The best way to understand these functions is to implement it yourself



HW 7-1: Background and hints

- Since async's can be called with any number of arguments, it should be variadic. C++ indicates this with an ellipsis "...". Look online or in our notes and homework for examples of where we've done this
- The return type, it is very messy, so we leverage the fact (mentioned in lecture 1) that C++ lets us leave the return type as auto and it will deduce it from the return statement
- This gives us a signature something like the following
 - ```
template<typename Func, typename ...Args>
auto my_async(Func f, Args... args) { /* ... */ }
```
- Since we don't do a join when using async (why?), you can call your new threads detach() method to allow it to run independently



# HW 7-2

- Use `std::promise` to create your own implementation of `std::packaged_task`



# HW 7-3

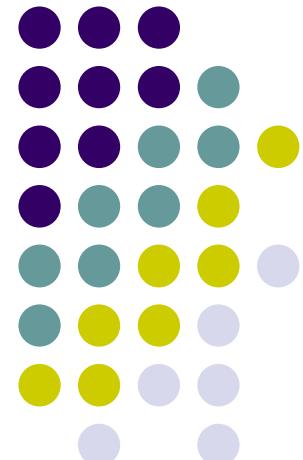
- Use a `std::condition_variable` to create your own versions of `promise` (again, only basic functionality required) and `future`
- Hint: To handle the case of passing an exception, look up `std::exception_ptr` and `rethrow_exception`
- Hint: Sometimes, esp. in multithreaded programs, an object may have multiple owners, so you need something other than `unique_ptr`, which models unique ownership. If you find you need that, look up `shared_ptr`, which models shared ownership using a reference count

# C++

## February 24, 2023

Mike Spertus

[spertus@uchicago.edu](mailto:spertus@uchicago.edu)



MASTERS PROGRAM IN  
COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO



# POLYMORPHIC CONTAINERS

# So far, our containers have only held objects of one type



- The main lecture will be about object-oriented design
- While object-orientation is all about polymorphism, all of the containers we have used hold objects of a single type
- `vector<int>` holds only integers
- `list<string>` holds strings
- `map<string, int>` lets you look up the integer corresponding to a string
  - `m[“foo”s] = 3;`



# We'll always have inheritance

- We can still use inheritance and virtual functions to put objects of different dynamic types in a container
- ```
vector<unique_ptr<Animal>> zoo;
zoo.push_back(make_unique<Gorilla>());
zoo.push_back(make_unique<Lobster>());
```

But there are times when you want different static types



- C++ also includes data structures that can hold objects of different types
 - These will come in very handy when we dig into object-oriented design later this lecture
- Let's see a few



pair and tuple

- Two of the most useful templates in the C++ standard library are pair and tuple
- They are to classes what lambdas are to functions
- Basically, they create an anonymous class on the fly
- Next quarter, we will create our own pair and tuple classes



Tuples

- A tuple is an anonymous struct with fields of given types
- You can think of tuples having the same relation to structs as lambdas do to functions
- You can use `get<0>` to access the fields
- ```
tuple<string, int, double> si("str", 2, 3.5);
tuple di(2.5, 3, 'c');
// CTAD: di will be a tuple<double,int, char>
cout << get<0>(di) // prints 2.5
cout << get<char>(di); // prints 'c'
int three = get<1>(di);
```



# Returning multiple values

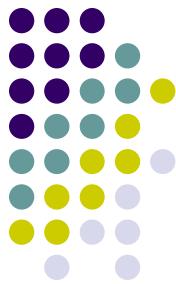
- One natural use for pair and tuple is to let functions return multiple values
  - ```
pair<int, int> f() {
    return {1, 2}; // ok
}
```
 - ```
tuple<int, int, char> g() {
 return {1, 2, 'u'}; // OK starting in C++17
}
```
- For more information on why it took until C++17 for the tuple example, see Improving Pair and Tuple (revision 1) by Daniel Krugler
  - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3739.html>
  - Warning: Very advanced



# Pairs

- Pairs are like tuples that always have two types
  - `pair<int, string> = { 1, "foo" };`
- Only difference is members also have name
  - `p.first` is effectively a synonym for `get<0>(p)`
  - `p.second` is effectively a synonym for `get<1>(p)`
  - Note that `get` is zero-based (as it should be) while the field names starts at one

# Why do we care about pairs if we have tuples?



- Now that there are tuples (C++11), there are no longer many reasons for you to use pairs (C++98)
- However, you will run into them
- For example, the element type of a map is a key value pair
  - ```
map<int, string> m = {{1, "foo"}, {0, "bar"}};
for(auto &kv: m) { // kv is a pair<int, string>
    cout << format("key: {}\tvalue: {}", p.first, get<1>(p));
```



Structured Bindings

- Structured binding are a feature that works very well with functions that return multiple values in a pair or a tuple



The problem

- Suppose you want to get several values back from a function
 - Use a tuple or a pair like we showed before
- But tuples are clumsy to work with. How do you get the values into individual variables where you want them after you've called the function?
 - The old way is to use std::tie
- ```
tuple<string, Rank, long> identity(Person);
string name;
Rank rank;
tie(name, rank, std::ignore) = identity(Mike);
```



# tie is a little bit clunky

- It can only assign to existing variables
- It doesn't deduce types
- It can't decompose types other than tuple/pair
- Can we improve?

# C++17 structured bindings (Examples from cppreference)



- ```
int a[2] = {1,2};  
auto [x,y] = a; // creates e[2], copies a into e, then x refers to e[0], y refers to e[1]  
auto& [xr, yr] = a; // xr refers to a[0], yr refers to a[1]
```
- ```
float x{}; char y{}; int z{};
tuple<float&,char&&,int> tpl(x,std::move(y),z);
const auto& [a,b,c] = tpl;
// a names a structured binding of type float& that refers to x
// b names a structured binding of type char&& that refers to y
// c names a structured binding of type const int that refers to the 3rd element of tpl
```
- ```
struct S { int x1 : 2; volatile double y1; };  
S f();  
const auto [x, y] = f(); // x is a const int lvalue identifying the 2-bit bit field  
// y is a const volatile double lvalue
```



More realistic example

- Suppose we have a function that returns either a web page or a network error code

```
• pair<unique_ptr<Page>, int> getWebPage() {  
    /* ... */  
    if(succeeded)  
        return { page, 0 };  
    else  
        return { unique_ptr<Page>, errCode };  
}  
/* ... */  
auto [ page, err ] = getWebPage();  
if(!err) processPage(page)  
else if (err == 404) { cout << "Not found" << endl; }  
else { cout << format("Unknown error {}\n", err); }
```



std::optional

- Sometimes it is useful for an object to be unset
 - E.g., like null in Java or None in Python
 - If you know about pointers, they can be null as well
- That is where std::optional comes in



std::optional example

```
// f may fail or return an int
optional<int> f()
{ return /* succeeded */ ? 1 : {} }

auto r = f();
if(r) { // f return a result
    cout << "succeeded: "
        << r.value() << endl;
} else {
    cout << "failed" << endl;
}
```



Variants

- C++ has a lightweight "variant" abstraction that generalizes C unions
- A variant<A, B> can hold an A or a B but not both
- These variants will be the basis of an approach to OO that will
 - Often have much better performance than virtuals
 - More dynamic than our Animal concept (we can have a zoo with runtime dispatch)
 - Everything is value-based, no need to worry about references, unique_ptr, RAII, ...
 - Great support for Open-Closed extensibility



Variants: Basic use

- Think of a variant is like a tuple, but instead of holding all of its fields at once, it only contains one of them at a time
- Supports a very similar interface to tuples
- ```
variant<int, double> v = 3; // Holds int
get<0>(v); // Returns 3
get<1>(v); // Throws std::bad_variant_access
v = 3.5; // Now holds a double
get<double>(v); // Returns 3.5
```
- You can also check what is in it  

```
v.index; // returns 1
holds_alternative<double>(v); // returns true
holds_alternative<int>(v); // returns false
```
- We will learn some more ways to access variants when we cover object-oriented design

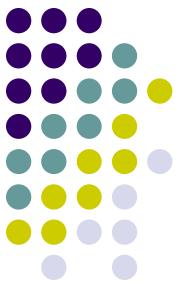


# "OO" DESIGN



# Why the quotation marks?

- C++ offers many ways to address the sort of problems covered by OO design
- Not all of these are considered "OO" techniques
- Let's look at an example



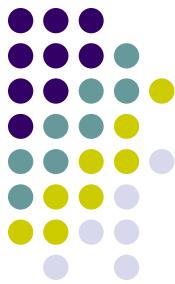
# TEMPLATES AND OO CAN SOLVE SIMILAR PROBLEMS



**Generic Programming  
Should “Just” Be  
Normal Programming**

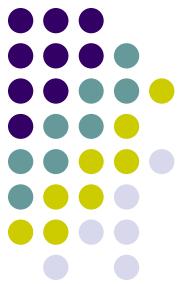
— Bjarne Stroustrup

# Using templates instead of inheritance and virtuals



- OO and templates can be used for many of the same problems
- In the spirit of Bjarne's quote, new C++ features, like C++20 Concepts, were intentionally designed to be familiar for replacing object-oriented code
- Let's review an example we looked at before

# OO and Templates can solve similar problems



- Consider the following OO code

```
struct Animal {
 virtual string name() = 0;
 virtual string eats() = 0;
};

class Cat : public Animal {
 string name() override { return "cat"; }
 string eats() override { return "delicious mice"; }
};
// More animals...

int main() {
 unique_ptr<Animal> a = make_unique<Cat>();
 cout << "A " << a->name() << " eats " << a->eats();
}
```



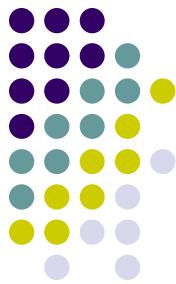
# Do we need to use OO?

- Not really

```
struct Cat {
 string eats() { return "delicious mice"; }
 string name() { return "cat"; }
};
// More animals...
```

```
int main() {
 auto a = Cat();
 cout << "A " << a.name() << " eats " << a.eats();
}
```

# That was a lot simpler but...



- We lost the understanding that `a` is an animal
- `a` could have the type `House` or `int` and we might not find out that something went wrong until much later when we did something that depends on `a` being an animal
- What we need is a way to codify our expectations for `a` without all of the overhead and complexity of creating a base class



# Concepts

- Concepts play the analogous role for generic programming that base classes do in object oriented programming
- A concept explains what operations a type supports
- The following concept encapsulates the same info as the base class

```
template<typename T>
concept Animal = requires(T a) {
 { a.eats() } -> convertible_to<string>;
 { a.name() } -> convertible_to<string>;
};
```

# Now, we can ensure that a represents an animal



- With the above concept defined, we can specify that a must satisfy the Animal concept, and the compiler will not let us initialize it with a non-Animal type like House or int

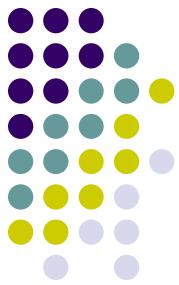
```
int main() {
 Animal auto a = Cat();
 cout << "A " << a.name() << " eats " << a.eats();
}
```



# Let's compare

- <https://godbolt.org/z/cWc6aM>
- As you can see, the definition of Cat and the client code in main() look very similar in both
- This follows a principle enunciated by Bjarne Stroustrup
  - “Generic Programming should just be Normal Programming”

# Usage is almost identical to before



```
Animal auto a = Cat();
cout << "A " << a.name()
 << " eats " << a.eats();
```



# How does this compare?

- Performance is better
  - Objects created on stack
  - No virtual dispatch
- No inheritance
  - Makes it easier to adapt classes to our code without risking “spaghetti inheritance”
  - On the other hand, it weakens type safety
    - Pacman is not an animal but eats and has a name
- No runtime polymorphism
  - The following is legal if Animal is a class but not if it is a concept (Why?)
  - `set<unique_ptr<Animal>> zoo;`



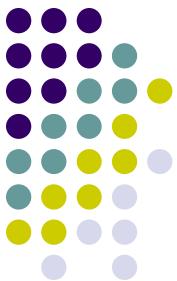
# A real world example

- Suppose C++ didn't have mutexes
  - It didn't until C++11
- How would we design them?
- Let's look at how Java does it
- Java uses inheritance and virtual methods



# Java-style mutexes

```
struct lockable {
 virtual void lock() = 0;
 virtual void unlock() = 0;
};
struct mutex: public lockable {
 void lock() override;
 void unlock() override;
};
struct lock_guard {
 lock_guard(lockable &m) : m(m) { m.lock(); }
 ~lock_guard() { m.unlock(); }
 lockable &m;
};
```



# Using our Java-style mutex

```
mutex m;

void f() {
 lock_guard lk(m);
 // do stuff
}
```

# Wait, that's not how C++ mutexes work!



- C++ mutexes do not inherit from a lockable base class
- The C++ committee decided to use templates instead of the virtual override approach taken by Java
- Since mutexes are frequently used in performance-critical code, this was undoubtedly the right choice
- Let's take a look



# C++-style mutexes

```
struct mutex {
 void lock();
 void unlock();
};
```

```
template<typename T>
struct lock_guard {
 lock_guard(T &m) : m(m) { m.lock(); }
 ~lock_guard() { m.unlock(); }
 T &m;
};
```



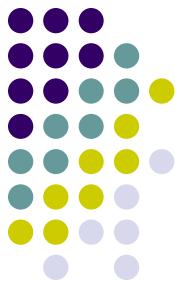
# Using our C++-style mutex is typically unchanged

```
// Exactly the same as before!
```

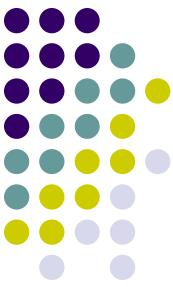
```
mutex m;
```

```
void f() {
 lock_guard lk(m);
 // do stuff
}
```

# Class Template Argument Deduction

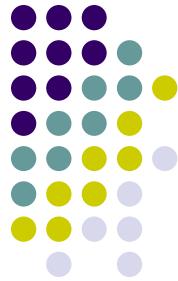


- Note the following line depended on C++17 CTAD
  - `lock_guard lk(m);`
- CTAD infers the template arguments for `lock_guard<mutex>` from the constructor similarly to how Function Template Argument Deduction infers the template arguments for function templates
- CTAD can often be useful in this way when using templates instead of virtuals. E.g., if `tp1` and `tp2` are of type `time_point<C, duration<R>>`
  - `duration d = tp1 - tp2; // duration<R>`



# So many choices :/

- As we saw above, there is usually a choice between base classes/virtuals and templates/concepts
- As we will see below, this is only the beginning
- C++ supports many approaches for "object-orientation"
- How can we make sense of this?
- We will need an understanding of OO design best practices



# **THE SOLID PRINCIPLES (H/T TONY VAN EERD)**

# Popularized by "Uncle Bob"

## Robert C Martin ~2004



- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle



# Single Responsibility Principle

- How often do you see a class that has members organized into subsets?

```
class Camera
{
 CameraId id; } // orange brace
 DevicePath path; } // orange brace

 int bitdepth; } // pink brace
 Resolution resolution; } // pink brace
 int gain; } // pink brace
 int exposure; // units? } // pink brace

 Pose pose; } // red brace
 Calibration * calibration; } // red brace

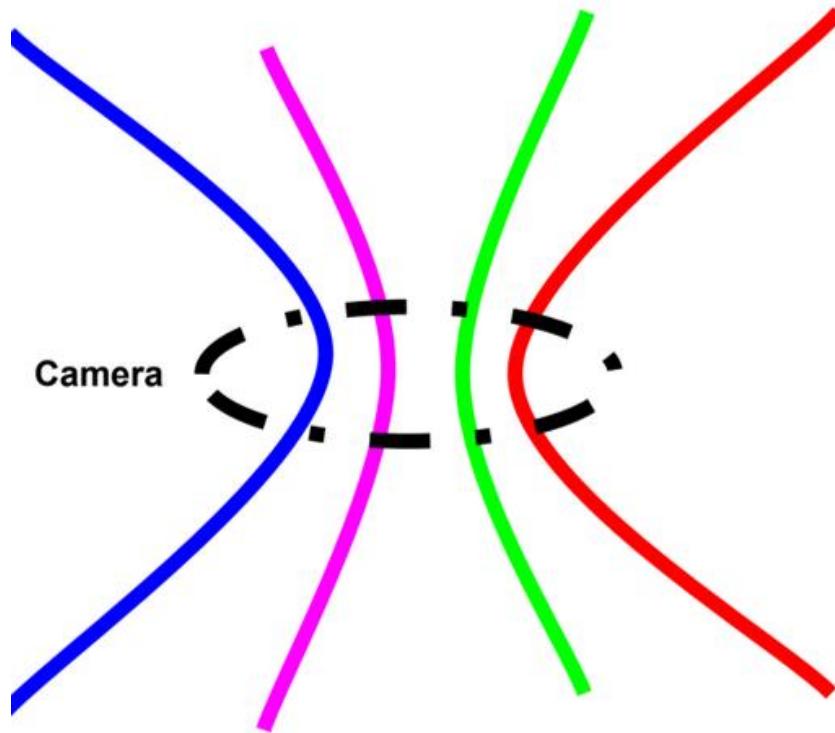
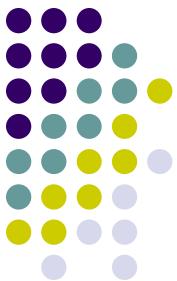
 int binarizationThreshold; } // blue brace
 int sharpness; // UI only } // blue brace
 int multicamEdgeThreshold; } // blue brace

 Image baseImage; } // green brace
 Image negativeImage; } // green brace
 Image maskToUse; } // green brace
 Image reverseMask; } // green brace

 Device * device; } // purple brace
 INativeCamera * camera; } // purple brace

 ImagePoller * poller; } // purple brace
};
```

These are separate pieces  
pulled together by association



# What if we wanted to use just one piece?



- Can I pose (position) things other than a camera?
- Does posing really have anything to do with resolution?
- What if a camera supported multiple resolutions but only a single pose?
- We can't do that with this class without breaking the Open-Closed principle
  - Which we'll learn about momentarily, but in this context it means "without breaking existing code"

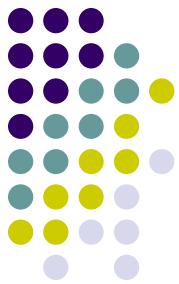


# DON'T CROSS THE STREAMS



memegenerator.net

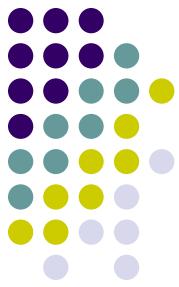
# Better: Give Image its own class



```
struct Image
{
 unsigned char * pixels;
 Format format; // RGB vs RGBA vs Gray vs ...
 int width;
 int height;

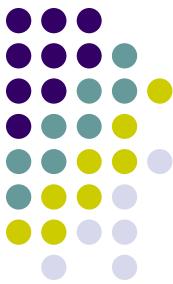
 Image();
 Image(int width, int height, Format format);
};
```

# It's not just classes that mix concerns

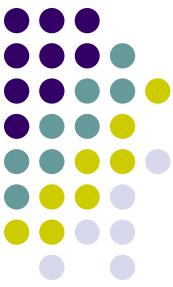


- What about functions?
- Many functions are thousands of lines long
  - E.g., <https://github.com/llvm/llvm-project/blob/b07aab8fc1088ef66ecbe2befc3ef7e3936a390e/clang/lib/Parse/ParseExprCXX.cpp#L154>
- ```
void myFunction(Quux q) {
    // Locals
    Pass p(1);
    Bar b(p);
    Blah l(q);
    // Setup the frabulator
    Frabulator f;
    f.x();
    f.y = "foo";
    // Loop through the sneetches
    for(auto s: f.sneetches)
        ...
}
```

How can I break a function into single-concern pieces?



- Breaking into several functions is typical
- But is sometimes difficult to share state
- ```
void myFunction(Quux q) {
 // Locals
 Pass p(1);
 Bar b(p);
 Blah l(q);
 Frabulator f = setupFrabulator(b, p, l, q);
 loopThroughSneetches(f, b, p, l, q);
 ...
}
```



# Option 1: More, smaller functions

- Breaking into several functions is typical
- But is sometimes difficult to share state
- ```
void myFunction(Quux q) {  
    // Locals  
    Pass p(1);  
    Bar b(p);  
    Blah l(q);  
    Frabulator f = setupFrabulator(b, p, l, q);  
    loopThroughSneetches(f, b, p, l, q);  
    ...  
}
```



Option 2: Use a functor

- Functions can be restructured as functors for better organization

```
• struct myFunctionHelper {  
    myFunctionHelper(Quux q) : q(q), l(q) {}  
    Quux q;  
    Pass p(1);  
    Bar b(p);  
    Blah l;  
    // Methods now have access to variables  
    Frabulator setupFrabulator() { ... }  
    void loopThroughSneetches() { ... }  
    int operator() {  
        Frabulator f = setupFrabulator();  
        loopThroughSneetches(f);  
        ...  
    };  
    int myFunction(Quux q) { return myFunctionHelper(q)(); }
```

- Good choice for really complex functions
- Supports powerful features like virtual functions and multiple entry points
- but high-friction for simpler cases

Option 3: Use lambda with capture lists



- While C++ doesn't have local functions *per se*, you can get the same organizing effect with local lambdas
- ```
int myFunction(Quux q) {
 Pass p(1);
 Bar b(p);
 Blah l;
 auto setupFrabulator = [&] { ... }
 auto loopThroughSneetches = [&](Frabulator f) { ... }
 Frabulator f = setupFrabulator();
 loopThroughSneetches(f);

 ...
};
```
- Good for "goldilocks" cases where a class is overkill but multiple responsibilities risk a "spaghetti" function



# Open/Closed Principle

- Software constructs should be open for extension but closed for modification
- Open for extension
  - Allows the addition of new capabilities over time
- Closed for modification
  - Don't break existing client code



# Inheritance and virtuals

- Of course, inheritance and virtual functions are a great way to extend classes
  - As are the template equivalents we discussed above
- Is it enough?
- Not quite
- It does not cover extremely common use cases for extensions

# The Problem: Client-side extension



- Suppose you are using a class hierarchy, and you wish the classes had a virtual method specific to the needs of your application
- Unfortunately, it probably doesn't because the class designer doesn't understand your application
- You may not be able to add them
  - Maybe they're not your classes
  - Maybe the virtuals you want only apply to your particular program, and it breaks encapsulation to clutter up a general interface with the particulars of every app that uses them



# The Visitor Pattern

- The Visitor Pattern is a way to make your class hierarchies extensible
- Suppose, as a user of the Animal class, I wished that it had a lifespan() method, but the class designer did not provide one
- We will fix that with the visitor pattern

# Class Creator: Make Your Classes Extensible



- Create a visitor class that can be overriden
- ```
struct AnimalVisitor {  
    virtual void visit(Cat &) const = Ø;  
    virtual void visit(Dog &) const = Ø;  
};
```
- Add an “accept” method to each class in the hierarchy
- ```
struct Animal {
 virtual void accept(AnimalVisitor const &v) = Ø;
};
struct Cat : public Animal {
 virtual void accept(AnimalVisitor const &v)
 { v.visit(*this); }
 /* ... */
};
```



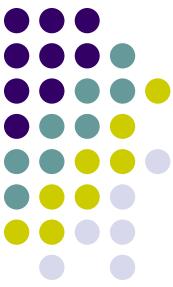
# Class User: Create a visitor

- Now, I create a visitor that implements the methods I wish were there
- struct LifeSpanVisitor  
: public AnimalVisitor {  
 LifeSpanVisitor(int &i) : i(i) {}  
 void visit(Dog &) const { i = 10; }  
 void visit(Cat &) const { i = 12; }  
 int &i;  
};



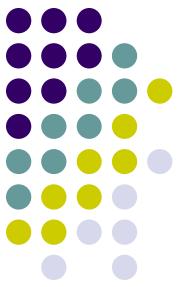
# Using the visitor

- Now, I can get the lifespan of the Animal a I created above
- int years;  
a->accept(LifeSpanVisitor(years));  
cout << "lives " << years;
- <https://godbolt.org/z/WbGe4z>



# Best practice

- If you are designing a class hierarchy where the best interface is unclear, add an accept() method as a customization point



# USING DUCK-TYPED VARIANTS FOR PERFORMANT, EXTENSIBLE OO



# Duck Typing





# Duck Typing

- OK. Not that
  - When I gave a talk on this in Shenzhen, I was worried what the translator would do with the last slide :/
- There is a saying

If it walks like a duck and quacks like a duck, then it is a duck

- Let's see if we can apply this to types



# Inheritance models “isA”

- As we've mentioned, inheritance is a model of the “isA” concept
- Duck typing gives a different notion of “isA”
- If a class has a walk() method and a quack() method, let's not worry about inheritance and call it a duck

If it walks like a duck and quacks like a duck, then it isA duck



# Templates use duck typing

- ```
T square(auto x) { return x*x; }
```
- `square(5); // OK`
- We don't require that the type of `x` inherits from a `HasMultiplication` class, simply that operator`*` makes sense to use here



Concepts

- C++20 Concepts improve duck typing
- We could have a HasMultiplication concept that would do the trick without requiring any complex inheritance hierarchies



Dynamic dispatch

- While C++ templates have always been duck typed, templates are used for compile-time dispatch
- By contrast, OO is used for dynamic dispatch
- Because duck typing is flexible and forgiving while remaining statically typesafe, people have asked whether we could use dynamically-dispatched duck typing as an alternative to inheritance

Using Duck-Typed Variants in place of OO



- Suppose we knew (at compile-time) all of the Animal classes
 - E.g., Cat and Dog
- However, we don't know the type of a particular animal until run-time
- Instead of inheriting from an abstract animal base class, we can have an animal variant
- using `Animal = variant<Cat, Dog>;`

How do I call a method on a variant?



- While `variant<Cat, Dog>` is a great way to store either a Cat or a Dog
- How can I simulate virtual functions and call the right `name()` method for the type it is holding?



The C++ standard library has a solution: std::visit

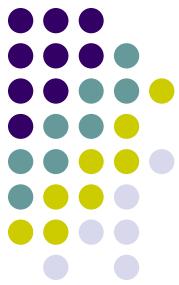
- **Warning:** Do not confuse with the Visitor Pattern we discussed earlier
- If v is a variant, and c is a callable, $\text{visit}(v, c)$ calls c with whatever is stored in v as its argument
- Does this solve our problem of making variants behave like virtuals?
- Let's see



Dynamically calling our Animal's name () method

- Animal a = Cat(); // a is a cat
- cout << visit([](auto &x) { x.name(); }, a);
- Prints “Cat”, just like we want
- How does it compare to using virtuals or templates?

In some ways, it's the best of both world



- Almost as fast as templates
 - Since Animal is a single type that can hold a Cat or a Dog, we can just use animals by value instead of having to do memory allocations
- As dynamic as traditional OO
 - A set<Animal> works great
 - Unlike our Concepts version



Malleable (mallard?) typing

- Virtual functions need to exactly match what they override, so we couldn't, say, give Cat's eat() a defaulted argument
 - ```
struct Cat : public Animal {
 void eat(string prey = "mouse") override; // ill-formed
```
- With variants, that is not a problem
- As long as eats() is callable, we don't care about the rest
- ```
struct Cat {  
    void eat(string prey = "mouse");  
};  
visit([](auto &x) { x.eats(); }, a); // OK. Eats a mouse
```



Users can add methods

- Just like we discussed with the Visitor Pattern, users of the Animal type can add their own methods
- To do this, we will use the “overloaded pattern”



The overloaded pattern

- Define an overloaded class (you only need to do this once)
- ```
template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };
```
- This inherits the function call operator of everything it is constructed with
- Let's make this clear by creating a lifeSpan() "method" like we did before
- ```
overloaded lifeSpan([](Cat &) { return 12 },
                     [](Dog &) { return 10});
// Get life span without knowing whether
// a is a Cat or a Dog
cout << visit(lifeSpan, a); // Prints 12
```
- Note that this idiom relies on CTAD and aggregates deducing the template arguments for you



Problems with Duck Typing

- The notation is much uglier than calling a virtual function
- While the flexibility is nice, duck typing reduces type safety
 - It cannot tell that a Shape's draw() method for drawing a picture is different than a Cowboy's draw() method for drawing a gun
- Variants always uses as much space as the biggest type
- Whenever we create a new kind of Animal, we have to add it to the variant, which can create maintenance problems



Best Practice

- Because it is ugly and not well-known, only prefer variant-based polymorphism over virtuals or templates when there is a clear benefit
 - In practice, I use it a lot, but less than I do virtual functions or templates



Liskov Substitution Principle

Subtype Requirement:

*Let $\Phi(x)$ be a property provable about objects x of type T .
Then $\Phi(y)$ should be true for objects y of type S
where S is a subtype of T*



Inheritance models "isA"

- Inheritance is one way of modeling subtyping
- A Deer isA Animal
- One would expect that anything that is true about animals is true about deer



Concepts model isA

- If Animal is a concept instead of a base class, we have seen that the same is true
- One other benefit of concepts is that inheritance only inherits methods, but concepts can specify almost arbitrary $\Phi(x)$ properties
- Tradeoff: Efficiency vs dynamism
 - Generally how to choose the approach



Tony says

Liskov Substitution Principle

```
struct Line
{
    explicit Line(LineSegment);
    explicit Line(Ray);
};
```

```
struct Ray
{
    explicit Ray(Line);
    explicit Ray(LineSegment);
};
```

```
struct LineSegment
{
    explicit LineSegment(Line);
    explicit LineSegment(Ray);
};
```

LSP

Litmus for explicit vs implicit

```
int func(Ray r);

int main()
{
    Line line = ...;
    return func(Ray(line));
}
```



Interface Segregation Principle

- No code should be forced to depend on methods it doesn't use
- (Martin) Suppose we have a fat "Job" class that has a bunch of methods that are only relevant to print jobs and other methods that are only relevant to stapling jobs
- If the stapling code takes a Job, it will needlessly only work with Jobs that also know about printing



Handling with OO

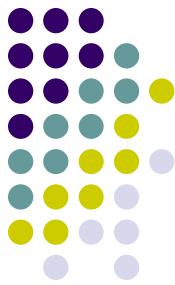
- This is often given as a motivation for using abstract base classes (interfaces)
- The concrete Job class implements the PrintJob and StapleJob interfaces
- This can be taken so far, getting into spaghetti inheritance and excessive multiple inheritance complexity

Concepts also handle this nicely



- The stapling code can only require what it needs to staple without exploding the type hierarchy
- However, you could also go too far with this too as an incoherent set of functions that each make different requirements of each job that is passed in
- Both of these are good reminders that architecture is more art than science

Dependency Inversion Principle



- This is sometimes paraphrased as "All programming problems can be solved with an extra layer of indirection"
- ☺



Basic idea

- “*the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions*”

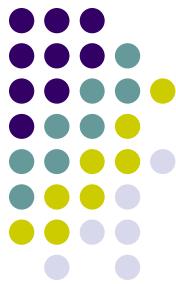


Example

- Suppose you have a thumbnail service class that looks for pictures in S3 folders

```
class ThumbnailService {  
    S3Folder inputFolder;  
  
    ...  
};
```
- It is now coupled with the concrete S3 service instead of an abstract idea of a storage service, which is probably sufficient for this use
- Again, the indirection can be introduced either through inheritance/virtuals or template/concepts
- Usual performance/dynamism tradeoff

Solving with inheritance and virtuals



- Class S3Folder : public Folder { ... };

```
class ThumbnailService {
    ThumbnailService(Folder &inputFolder)
        : inputFolder(inputFolder) {}
    Folder &inputFolder;
    ...
};
```

Solving with templates and concepts



- `template<Folder F> // Folder is a concept`
`class ThumbnailService {`
 `Folder &inputFolder;`
 `...`
`};`
`ThumbnailService<S3Folder> ts;`
`ThumbnailService ts2(myS3Folder); // CTAD`

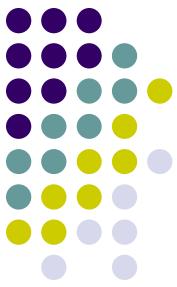


TYPE ERASURE



Type Erasure

- One would often like to use templates as described in this lecture to work with different types
- But your use case needs actual classes rather than templates
 - We've already discussed wanting a fixed type to put in a vector
 - We'll see some more examples below
- Type erasure allows you to store a variety of types in a (non-template) class



Example: std::any

- std::any
 - Uses type erasure to store anything at any time
 - Use any_cast to get the value out
 - bad_any_cast exception if you try to take out a different type
 - any a = 7;
`cout << any_cast<int>(a);`
a = "foo"s;
`cout << any_cast<string>(a);`



Example std::function

- We have discussed how `std::function<int(double)>` can contain any callable (function, functor, lambda,...) that accepts a double and returns an int
- How can it store something of variable type
- By erasing the type
- Let's see how it works

Class, Concept, and Model (Rob Douglas)



- There are three parts to Type Erasure
- The class
- The concept
- The model



The class

- This is the non-template class you want to be able to hold various types
- Typically it does have a constructor template or a template assignment operator to let you store various types in it
- ```
struct Erased {
 template<typename T> Erased(T&&);
 template<typename T>
 Erased& operator=(T&&);
};
```



# The Concept

- You might not want to store arbitrary types
- The concept explains what interface you want to support
  - Similar (but not identical) to C++20 concepts
- Declare a nested Concept class with a pure virtual interface
- ```
struct Erased {  
    template<typename T> Erased(T&&);  
    template<typename T>  
    Erased& operator=(T&&);  
struct Concept { virtual void foo() = 0; };  
};
```



The model

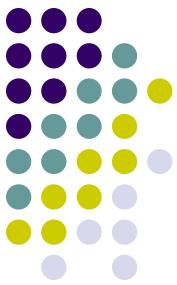
- Class template that stores the actual object (of unknown type) and delegates calls

- struct Erased {
 template<typename T> Erased(T&&);
 template<typename T>
 Erased& operator=(T&&);

```
struct Concept { virtual void foo() = 0; };
```

```
template<typename T>
struct Model : Concept {
    T val;
    Model(T&& val) : val(val) {}
    void foo() override { val.foo(); }
};
```

Store and access the type-erased field



```
struct Erased {  
    template<typename T> Erased(T&& t)  
        : valp(make_unique<Model<T>>(t)) {}  
    template<typename T>  
    Erased& operator=(T&& t)  
        { val = make_unique<Model<T>>(t); }  
  
    struct Concept { virtual void foo() = 0; };  
  
    template<typename T>  
    struct Model : Concept {  
        T val;  
        Model(T&& val) : val(val) {}  
        void foo() override { val.foo(); }  
    };  
    void foo() { val->foo(); }  
    unique_ptr<Concept> val;  
};
```



Let's use it

```
struct A { void foo(); };
struct B { void foo(); };
```

```
int main()
{
    Erased e(A());
    e.foo(); // Calls A's foo
    e = B();
    e.foo(); // Calls B's foo
}
```

Note: This works even though A::foo and B::foo are not virtual (the virtualness is in the Model and the Concept)



HW 8-1

- *Static polymorphism* is another common idiom for using templates to achieve "OO-like" behavior at compile-time
- Read the static polymorphism section of <https://iamsorush.com/posts/static-polymorphism-cpp/>
- Reimplement our "animal example" from <https://godbolt.org/z/cWc6aM> using static polymorphism
- What is your opinion of this approach? Do the SOLID principles shed any light on that?



HW 8-2

- Choose some C++ program (e.g., from GitHub)
- Analyze it according to the SOLID principles
- What does it do well?
- What does it do badly?
- Can you suggest improvements?