Learning outcomes:

- problem-solving approach
- practical language skills

In this course, we’ll consider both the intellectual and the practical sides of computer programming.

Programming is __________________________. It’s the application of an intellectual approach to practical problem-solving (which is actually a pretty serviceable definition of __________________________!).

We also, as developers of real software, need to be familiar with __________________________. We’ll do this too. As a side benefit, note that a solid grasp of C++ is a highly marketable skill.
Problem solving

- Intellectual content
- Language-agnostic
- Broadly applicable

- explain and differentiate fundamental abstractions and types,
- read and understand well-written programs,
- evaluate the correctness of pseudocode and C++,
- diagnose common errors in memory management and simple algorithms,
- design simple object-oriented systems; and
- synthesize correct and idiomatic C++ with simple STL containers.

All of the important ideas in this course can apply to many programming languages — or none at all! In fact, an awful lot of the material in this programming course doesn’t have to have be applied to computers. A study of abstractions, algorithms and types can be ____________ ___________ and ________________________ to logistics, writing procedures and manuals, etc.

However, we will focus on computer programming. This isn’t a C++ course: it’s a foundational programming course. This slide highlights the aspects of the course outcomes that aren’t necessarily C++-specific, and it’s most of them.
Practical skills

- C++
- Guided exploration: labs
- Practice: problems, assignments

- explain and differentiate fundamental abstractions and types,
- read and understand well-written programs,
- evaluate the correctness of pseudocode and C++,
- diagnose common errors in memory management and simple algorithms,
- design simple object-oriented systems; and
- synthesize correct and idiomatic C++ with simple STL containers.

However, we can’t teach ideas in a vacuum. Often, the best way to ____________ is to ____________, and in this course we will apply ideas using the C++ programming language.

We will also have some practical, hands-on labs to guide you through the exploration of real running software and important software development tools.

Throughout the course I will provide problems and exercises for your own personal practice. The assignments are meant to do the same thing: encourage practice. I would strongly suggest that you take any practice opportunity you can get.
How to succeed

Ask questions

Practice!

- exercises from lectures and tutorials
- practice problems from textbook(s)
- problems inspired by other courses

If you’re here, ________________. Getting through Engineering One means that you have all the ability you need to pass this course.

However, ________________. I will do my part; I will attempt to make the material as ________________ as possible and give you the ________________ you need to succeed. However, you must take ________________ of your progress.

I have found that the students who practice programming tend to succeed. This doesn’t have to require a lot of time. I challenge you to spend ________________ ________________ working through exercises and problems. It’s only when you ________________ knowledge that you find out whether or not you really ________________ it.
How to succeed

Components

• Lectures
• Labs
• Tutorials

Labs

Unlike ENGI 1020, our labs have nothing to do with assignments. Instead, they're more like traditional Engineering labs: ______________ of course material and exposure to ______________.

Tutorials

We'll use the tutorial slot for different kinds of things in different weeks: set exercises, unstructured question-asking, etc. Worth your while.

In the first week, I'll host a ______________ to help you get a decent C++ compiler running on your own computer. It's not required for the course, but it's highly recommended.
How to succeed

Communication:

- Office hours (10-11am, Monday and Friday)
- Appointments
- D2L forums
- Course website

Office policy: my door is usually open — if it is, come knock!
This course doesn’t follow a textbook: it’s about preparing you for work terms and further courses in the ECE program. However, I do recommend buying a textbook as a reference and source of further example problems.

You might already have Friedman & Koffman, but I really do recommend Stroustrup for its use of modern C++ features. It’s also one of the cheapest textbooks you’ll ever buy --- you can get it in paperback for $65 or Kindle Edition for $47. There’s no reason not to get this book.
The University has standards of honesty and integrity for all its students, but you are also
_______________________________ with ethical obligations. When you complete a piece of
evaluation, you must be confident that it was done ethically: it must be ____________________
______.

Helping each other learn is important. Help each other understand the material and the
assignments, but when you put your name on a submission, you are saying, _________________.
Think about ___________________: if you come to me with questions, I will:

1. talk through the problem with you,
2. ensure that you understand the relevant concepts,
3. ask you questions about the assignment and your approach and even
4. ask you, "what next? and then?".

That process is intended to be helpful, but when you leave my office, I will have helped you find the
solution, not given you the solution. I will certainly __________ let you look at my solution! Your
interactions with each other over labs and assignments should work like that: ________________
______.
Topic 1:

Abstractions and programming paradigms
Two views:

- Mathematical

\[ f(x) = \begin{cases} 
  x, & x \geq 0 \\
  y, & x < 0 
\end{cases} \]

- Abstract machine

The mathematical view of the world is extremely abstract: it __________ everything until we reach a problem that can be solved mathematically. If you’ve heard the joke about __________ __________ you’ll know what I mean by generalization.

Another view is the classic Engineer’s view: "give me a machine that I can tinker with". In this view, we have bits that are organized in memory and we have systems for manipulating them and interacting with peripheral components. This is the foundation for computation, but we’re so deep in detail, it’s easy to ____________

So, which approach is the best? It depends on the problem! That is, it depends on ________ ________ we need to be. Right now, I need to understand that each of you is a __________ __________. When you come to my office hour, however, I need to understand more about you as an individual: what you’re good at, struggling with, etc. That level of detail would be ____________ in a large lecture like this.

Fortunately, we don’t have to choose between just two approaches...
Abstractions can be found on a spectrum, from the very abstract at the top (of this slide) to the very practical at the bottom. Sometimes, the best answer is an equation. Sometimes, it's a circuit. In between, we have many different kinds of programming. Programming can be done quite abstractly (almost pure math) or in a very detail-oriented way (turning individual bits on and off). One of the reasons we're using C++ is because ______________________________.
Programming languages

- To communicate with computers
- To communicate with people
  - Other developers
  - Test/verification teams
  - Financial traders
  - Yourself!

Most people understand that programming languages are used for communicating with computers: for giving them instructions. What often isn’t appreciated is that programming languages are also about communicating with... ____________!

People will spend more time reading your code than compilers. Other developers will need to understand what you’ve done and ____________________________. Other software development professionals will need to be able to understand what your code does and what it was supposed to do.

Seriously! In 2010, the US SEC (Securities and Exchange Commission) proposed that vendors of Asset-Backed Securities (think of subprime mortgages) have to produce a Python program that describes their crazy-complex financial instruments:

We are proposing to require that most ABS issuers file a computer program that gives effect to the flow of funds, or “waterfall,” provisions of the transaction. We are proposing that the computer program be filed on EDGAR in the form of downloadable source code in Python... (page 205)

Under the proposed requirement, the filed source code, when downloaded and run by an investor, must provide the user with the ability to programmatically input the user’s own assumptions regarding the future performance and cash flows from the pool assets, including but not limited to assumptions about future interest rates, default rates, prepayment speeds, ... (page 210)

(see https://jrvarma.wordpress.com/2010/04/16/the-sec-and-the-python)
Programming style

Not "style", style:

When we talk about programming "style", we're (hopefully) not worried about fashion. Rather, we're talking about conventions that promote ________________________.

When a news organization produces a style guide, they're not telling their journalists how to dress, they're laying down rules for how "we" use punctuation and grammar, how "we" attribute sources, how "we" use controversial terms. That is, when you read the CBC, the Guardian or the New York Times, you should see internally-consistent meanings of terms like "activist" or "private military contractor". This makes the content ________________________.
There are many ways to think about computer programming, but we’re going to look briefly at four of the main ones and then spend most of the term on the first three.
Procedural programming should already feel familiar to you: it's what you did in ENGI 1020 or your first tinker with Basic or JavaScript. Procedural programming is about ___________ and then ____________.

For example, the `drive` function on this slide might make a robot drive a fixed distance forwards or backwards. If you wanted to write a procedure for solving a maze with a robot, you wouldn't want to repeat the "drive forward" procedure over and over — instead you'd use the procedure that's already been given. This is also true of non-computer programming, like an instruction manual.

One aspect of this programming style is that it's imperative: it consists of lots of commands that ___________.
### Functional programming

- Using functions as variables:

```scala
def triple(x: Int) = 3 * x
val some_numbers = List(1, 2, 3)
val bigger_numbers = x.map(triple)
```

- Popular in 1970s (e.g., Lisp), coming back:
  - Haskell, Scala, OCaml
  - C++!

---

Functional programming is more about _________________ than the details of ________________.

C++ acquired some really nice functional primitives in the 2011 version of the standard — we’ll come back to these towards the end of the course.

Part of C++’s strength as a language is that it is multi-paradigm: it’s so flexible that you can do just about any kind of programming in it. The downside of this flexibility is that people do try to do every kind of programming in C++, which can make for difficult-to-read code. A critical part of any company or project style guide is telling developers what language features are or are not used.
Object-oriented programming is centred around **objects**, which are bundles of ____________
__________________________.

For instance, in this example we can have different type of `Shape` object, each of which knows how to calculate its own area. A `Circle` object will contain a radius and some code that knows how to calculate the area from it \((2\pi r)\), whereas a `Rectangle` object will contain a length and width, together with its own area computation code \((l \times h)\).

You’ve already seen one example of an object type in ENGI 1020: `std::string`. However, they can get much more interesting and powerful than that! We will spend a lot of our time this term on object-oriented programming.
Declare things to be true:

\[
\begin{align*}
arc(a,b). & \quad arc(b,c). \\
arc(a,c). & \quad arc(a,d). \\
arc(b,e). & \quad arc(e,f). \\
arc(b,f). & \quad arc(f,g).
\end{align*}
\]

Query the program for facts:

?- arc(a,X). /* arcs from node a got to node X */
X = b
X = c
X = d

Finally, logic programming is a way of reasoning about true statements. In logic programming, we don’t tell the computer how to figure out whether or not there is an arc from a to any other point, we just program it with facts and enough information about relationships that it can go off and search all of its knowledge for a solution.

In the 1970s-1990s, AI researchers got really excited (and maybe still are?) about this kind of thing, convinced that we’d replace doctors with computers. They’re partially right (people do use WebMD), but not as much as they’d hoped. It turns out that people are still useful and hard to replicate with computer programs.
Summary

- Abstraction spectrum
- Programming languages
- Programming paradigms
Why C++?

- General-purpose
- Multi-paradigm(ish)
- Efficient
- Employable
- Foundational

C++ is a very flexible language. Many programming languages impose rules that in C++ are matters of style and convention. If you can imagine some behaviour for a computer, you can probably write the code in C++. You will be able to write your ideas, and understanding C++ will help you implement your ideas.

Thanks to C++’s flexibility, you can write procedural, object-oriented, functional (more and more) or just about any other kind of program you might want.

The implementers of C++ are relentless about it being efficient. Other languages may trade speed for safety, but not C++. This makes it a useful language to implement other languages in.

Lots of companies use C++ or languages that are easier to learn once you’ve learned C++. Also, if you’re a C++ master, you can go work at Apple, Facebook, Google, NASA...

If you’re a pretty good C++ programmer, you can learn pretty quickly (in fact, you can do this in ENGI 5895 — we don’t offer a Java course, we just highlight some key differences from C++). Or Python. Or Go. Or Rust. Almost every new programming language describes itself as a C++ killer. Understanding C++ gets you down the road to all of these.
C++11 was a big change for the language. After years of stagnation, the standard brought in fundamental new techniques that made the language much better.

Some of the biggest changes in C++11 have to do with ____________: features that you can use to prevent yourself (or those who work for you) from making mistakes. These are incredibly important, and we’ll spend time on them later in the term.

It made a lot of things ________________, but also introduced some _______ ____ ___________ and usually _______ _______ _______ because of the ________________: C++98 code needs to mostly "just work" in a C++11 environment.

Legacy support makes things complicated. There are ways in which this flexible language is too complex (very, very few people actually understand the whole thing), but you can stick to the simpler parts that everyone actually uses and do whatever you need (e.g., get a job!).
Topic 2:

Types
int i = 42;
double x = 3.1415926;
string s = “Hello, world!”;

Questions:

• what do these types tell us?

• how have you used these types?

• when were types created?

The answer to the last question might be surprising to you: types were first invented in _________, long before electronic computers existed (this was also the year of the ________ ____________, depicted on the slide). This most fundamental of computing ideas wasn’t originally a computing idea at all! Rather, it came from the world of ________________ ____________.
Set theory:

\[ S = \{ x \mid \frac{x}{2} \in \mathbb{Z} \} \]

Russell's paradox:

Let \( R = \{ x \mid x \not\in x \} \)

then \( R \in R \iff R \not\in R \)

Paradox!

_______________ was a philosopher in the early 20th Century who did some thinking about mathematical set theory.

Recall that one can define mathematical sets using the notation pictured here. In the first example, we define the set \( S \) to be the set of all values of \( x \) in which \( \frac{x}{2} \) is a member of the integers. That is, \( S \) is ________________. However, sets can contain things other than numbers: for instance, other sets.

Russell posed a problem to set theorists. He proposed a set \( R \) that contained _______________. This is very simple to write in set notation. However, there is a very practical problem!

Let's assume that _______________. In that case, \( R \) is a set that _______________. Therefore, it ought to be _______________. However, if \( R \) does contain itself, then it _______________! This is a paradox.
This is a paradox in the same sense that Doctor Emmett Brown would use the word:

- Marty goes back in time
- Marty stops his parents from falling in love
- Marty is never born
- if Marty doesn’t exist, there is nobody to go back in time and stop his parents from falling in love!
Type theory

\[ S = \{ x | \frac{x}{2} \in \mathbb{Z} \} \]

where \( S \) is a set of numbers

Now:

\[ R = \{ x | x \notin x \} \]

where \( R \) is a set of... what?

Invalid equation

We can resolve the paradox by introducing type theory. Originally, in the untyped set theory, we defined sets to be things that contain things.

In the typed set theory, we add a constraint to \( S \): we specify that \( S \) is a ________________.

Now, when we try to re-create Russell’s Paradox, we can’t get there. We cannot write down a valid typed equation that causes the paradox.

This resolves the paradox, because ________________ ________________ ________________ ________________. Writing that equation in the typed world order would be like writing \( 1 = 2 \): untrue, but not in any Universe-shattering paradoxical way.
Abstract machine:

\[ a = \text{true} \]
\[ b = 42 > x \]
\[ z = x + y \]
\[ c = a \text{ or } b \]
\[ \text{goto } z \text{ if } c \]

So, that’s a neat and tidy answer to how we can _____________. What’s the relevance to _____________?

Recall our abstract machine representation of a computer. In particular, what is the relevance of types to the _____________ and _____________?
Here is a depiction of some memory. Memory contains 0’s and 1’s. But what do they mean?

If the memory is interpreted as a 32-bit integer, the meaning is 1,078,530,010:

\[2^{30} + 2^{22} + 2^{19} + 2^{16} + 2^{11} + 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 2^3 + 2^1\]

However, if this is a 32-bit floating-point number, then the same pattern of bits represents an entirely different number. If the computer represents all numbers as 1’s and 0’s in memory, ______
________________________ what these bits are supposed to mean. It keeps track of the bits, but __________________________.
To get even more practical, let's look at a couple of examples from a more realistic processor (not an abstract machine).

All of your notebook computers, desktop computers, etc., have processors that use some variant of the x86 instruction set (named for the 8086, 286, 386, 486, etc.). The instruction set is the _______ ______________________ with a processor: a program (such as a C++ program) is translated by the compiler into a ______________________ taken from this set.

Some example instructions are shown on this slide: we can multiply unsigned integers, multiply signed integers, multiply floating-point numbers and more. Depending on what type of number we claim to pass in, we will get very ______________________ out (e.g., 1,078,530,010 vs $\pi$).

However, we are now getting off into the weeds. You will see assembly language next term; let us now go back to our abstract machine representation.
In our abstract machine representation of a computer, memory contains a vast ___________ ______, not identifiable, typed variables. When the CPU performs operations on bits from memory, it does whatever operation the program tells it to: it is the responsibility of the _________________ and the _________________ to keep track of what types should be used to interpret each region of memory.
Basic types:

```cpp
bool b = true;
char c = '?';
int n = 1000000;
double x = 3.1415926;
string name = "Jonathan Anderson";
```

C and C++ have a number of basic types that can be directly represented by 1’s and 0’s in memory, and which most processors can directly manipulate. These include integer and floating-point numbers, which can also be used to represent other things (e.g., a number between 0 and 255 can be used to represent characters in the ASCII alphabet).

On most notebook and desktop computers, there are a few standard sizes for different kinds of numeric value:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 B</td>
</tr>
<tr>
<td>char</td>
<td>1 B</td>
</tr>
<tr>
<td>int</td>
<td>4 B</td>
</tr>
<tr>
<td>float</td>
<td>4 B</td>
</tr>
<tr>
<td>double</td>
<td>8 B</td>
</tr>
</tbody>
</table>

More complex types such as string can be made from __________ containing "sub-variables" with their own representations and operations.
Specific integers

```c
#include <stdint.h>

// Signed integers:
int8_t a = 127;                   // 0x7f
int16_t a = 32767;                // 0x7fff
int32_t a = 2147483647;           // 0x7fffffff
int64_t a = 9223372036854775807;  // 0x7fffffffffffffff

// Unsigned integers:
uint8_t a = 255;                  // 0xff
uint16_t a = 65535;               // 0xffff
uint32_t a = 4294967295;          // 0xffffffff
uint64_t a = 18446744073709551615;// 0xffffffffffffffff
```

In addition to the platform default sizes for `char`, `int`, etc., C/C++ allows you to specify precisely __________ an integer representation you want to use.

This slide shows some specific values for different types of integer. For signed integers, the largest possible positive value is a zero followed by $n - 1$ ones, which is $2^{n-1} - 1$ (where $n$ is the total number of bits in the integer representation). For unsigned integers, the largest possible value is $2^n - 1$ (since we start counting at 0).
When we compile a C++ program, then, the compiler converts our code’s typed variables into ________________ that manipulate raw bits. The compiler must ensure that it provides the processor with the instructions to manipulate the right size and type of value: on its own, ________________ what type the variable has.
More complicated types

```c
bool b = true;
char c = '?';
int n = 1000000;
double x = 3.1415926;
string name = "Jonathan Anderson";
```

What's in a name?

- some characters and an integer length
  (roughly!)

string is a class that defines a new type

So how does the processor understand string? ________________

A string variable is an object made up of other "sub-variables", each of which has its own type (and therefore its own representation). Creating these kinds of more complex types is about 75% of this course!

In the case of string, each string variable will have its own ________________ as well as a ______________ representation.
Types, then, are about two things:

**Representation**

How the bit-representation of a variable in memory should be interpreted. In ENGI1020 we said that types have sizes: this was ______________________. The bigger picture is that types have representations, which do say how big a variable is, but also ______________________

**Operations**

The things that ______________________ variables of this type. We said in ENGI1020 that it wouldn't make much sense to subtract strings, so "that's not allowed". In this course, we'll see how and why different operators are applied to types.
The ________________ are binary operators (operators with two operands) that evaluate to **true** or **false**. The relationships that they test for are mostly the same as their mathematical relatives, although we have to be careful with equality: the equality operator is used to ______________ for equality between two expressions. When we want to ______________ two things equal, we need to use the **assignment operator** (=).
Operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Applies to</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
<td>int, double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Concatenate string</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
<td>int, double</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
<td>int, double</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
<td>int, double</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>int</td>
</tr>
</tbody>
</table>

These binary operators are a little bit more interesting: instead of always evaluating to true or false, they evaluate to something whose type _________________.

Here we also see that one operator (+) can mean different operations when applied to different types (addition vs concatenation, the joining of strings into a longer string).
### Operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Applies to</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>Add &amp; assign</td>
<td>int, double</td>
</tr>
<tr>
<td></td>
<td>Append</td>
<td>string</td>
</tr>
<tr>
<td>-=</td>
<td>Subtract &amp; assign</td>
<td>int, double</td>
</tr>
<tr>
<td>*=</td>
<td>Multiply &amp; assign</td>
<td>int, double</td>
</tr>
<tr>
<td>++</td>
<td>Increment</td>
<td>int, double, iterators, ...</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
<td>int, double, iterators, ...</td>
</tr>
</tbody>
</table>

The "in-place" operators also have type-specific operations. We are also now introducing the idea that ___________ ++ -- ___________________ like iterators (which we will spend time studying later in the term). In fact, we can ___________ ___________ for operators when we create types. This will be an important part of what we do this term.
Given:

```c
int a = 42;
int b = 97;
char c = '!';
double x = 3.14;
string s = "Hello, world!";
```

Try:

```c
a + b;  // Does this work? What does it mean?
s / a;  // What compiler error do you see?
b % a;  // What is this operation?
x % a;  // What happens here?
s + a;  // What is the result?
```

An exercise for the reader: given these five variables (e.g., defined at the top of a function), what should we think of the five expressions at the bottom of this slide?

1. What are the types of each operation’s operands?
2. Is there an operation defined for this operator and these operand types?
3. Which of these expressions compile?
4. What do the valid ones evaluate to?

Try reading these expressions and answering the questions above without using the compiler. Then, attempt to compile the code and compare the results to what you expected.

*Hint*: the given expressions, on their own, won’t print out any values for you to see — you will need to alter the code a bit in order to get useful results out.
(Very) rough definition:

*Something that can be evaluated: we can turn it into a value (which has a type!)*

\[
\mathbf{\nabla} \times \mathbf{E} \\
\frac{\partial \mathbf{B}}{\partial t} \\
\begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix} = \begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix}
\]

Mathematical expressions, such as those pictured here, have something in common: they can be ____________, or turned into values.

Each of these expressions, then, evaluates to a value. That value might be simple (like an integer) or complicated (like a matrix), but it's still __________ value, and we can describe its type. For instance, the curl of the electric field \((\mathbf{\nabla} \times \mathbf{E})\) is a three-dimensional vector with units T/s (teslas per second).
In this sense, programming language expressions (in C++ or any other language) are not so different from mathematical expressions. Every expression can be ______________ to a typed value, which may be simple or complicated, but is still __________ value.

The first expression on this slide evaluates to the integer 4. The second expression is more complex, but it too evaluates to a single number: a double-precision floating-point number approximately equal to 0 ("approximate" because 3.1415926 is only approximately equal to π). The third expression is interesting in a different way: it is accessing an element within an array of values, but it too evaluates to a single value. Finally, the fourth expression evaluates to something called an object, which we’ll explore later in the course, but which we will currently say is a ______ _________ that ______________________ (in this example, an array of characters that spell "Hello, world!" and the value length = 13).

You can explore some of these expressions by modifying the expression-types.cpp code example.
"Simple" addition:

```
1 + 3.1415926  // int + double
```

Not so simple!

When we looked at the `expression-types.cpp` example, we saw that the result of adding an integer and a double-precision floating-point number together was a `double`. This might seem obvious if we think about a pen-and-paper calculation (something without a decimal point plus something with one is likely to end up with a decimal point), but in fact, the case is not so simple... our abstract machine can’t execute an instruction like this.

Our abstract machine model of the computer, which is based on the capabilities of real, physical CPUs, simply doesn’t have such an instruction. We can add an `int` to an `int` or a `double` to a `double`, but not an `int` to a `double!`
Machine can add:

\[
\text{int} + \text{int} \\
\text{double} + \text{double}
\]

How can we add:

\[
\text{int} + \text{double}
\]

Given that our abstract machine can add an \text{int} to an \text{int} or a \text{double} to a \text{double}, we need to do something else if we're going to be able to add an \text{int} to a \text{double} like C++ allows us to.

____________ not all programming languages allow this kind of playing fast and loose with types. For instance, the OCaml programming language (used in all kinds of interesting places, including high-frequency financial trading firms) makes a clear distinction between integer and floating-point addition, disallowing the kind of \text{int} + \text{double} expressions that are permitted by C++, Java, Python, Rust, Go...
Type conversions

- Compiler will try to find a way
- `A op B`: try to convert `A` or `B`
- Some conversions are safe

When we encounter this kind of expression in C or C++ (or in most other programming languages), the compiler will try to find a way of doing what the programmer is asking. This adds a little work for the compiler to do, but it makes the language a bit easier (and quicker!) to write.

The compiler will see if an operand (value being operated on by an operation) can undergo type conversion to satisfy the operation. The language specifies a set of type conversions that the compiler is allowed to employ automatically, with no explicit programmer intervention — these are called implicit conversions.

Some — but ________________ — of these implicit conversions are "safe", meaning they will always preserve all of the information they're given. We will look at three kinds of implicit conversion:

- numeric promotion
- numeric conversion
- boolean tests
Safe!

```c
sizeof(bool) <= sizeof(char) <= sizeof(int) <= ...
```

```c
// All of these implicit conversions are safe:
bool b = true;
char c = b;
int i = b;
long l = i;
```

__________ sizeof is a C/C++ keyword that means, "Dear compiler, please figure out how many bytes we would need to represent the value that this expression evaluates to". It looks like a regular function call, but it is actually evaluated at __________ rather than _______ ________. We’ll get more precise about kind of thing later in the term.

C and C++ don’t specify how many you need to store, e.g., an integer: int might be 32b on your notebook computer, but it’s only 8b on a 3π robot. What __________ specified is that a bool can be __________ than a char, which can be no larger than an int, etc.

This chain of inequalities means that conversion from a bool to a char, a char to an int, etc., is always safe: it never loses information. We can always move a small thing to a bigger box without doing violence to the small thing (going the other way, however, might require breaking the thing into pieces). If we store the number 6 in an 8b space (0000 0110), it’s perfectly safe to move it to a 16b space (0000 0000 0000 0110) or a 32b space (0000 0000 0000 0000 0000 0000 0110). This is demonstrated in the numeric-promotion.cpp ________ ________.
Floating-point too:

```c
float f = 3.1415;
double d = f;
```

The same is true of floating-type types: a single-precision floating-point number (`float`) has fewer bits than a double-precision floating-point number (`double`), so it's always safe to store a small or not-terribly-precise value in a representation that can handle huge or super-precise values.
Unsafe conversions also allowed!

```c
long bigNumber = (1 << 33) + 5;
int i = bigNumber;

unsigned int huge = (1 << 31);
int signed = huge;
```

```
unsafe-conversions.cpp:8:22: warning: implicit conversion changes signedness: 'unsigned int' to 'int'
    int signedVersion = largeInteger;
               ^~~~~~~~~~~

unsafe-conversions.cpp:13:12: warning: implicit conversion loses floating-point precision: 'double'
    float f = d;
       ^

unsafe-conversions.cpp:20:18: warning: implicit conversion turns floating-point number into integer
    int almost_pi = pi;
           ^

3 warnings generated.
```

In addition to "safe" numeric promotion, the compiler will also implicitly convert some values in a way that can ________________ . For example, the long integer type on most computers can hold bigger numbers than the int type: if you were to try and put 64b of integer into a 32b space, then you will lose information about the original number. You simply ___________ ______ accurately. And yet that's exactly what the compiler will do!

________________________ unsafe-conversions.cpp

The good news is that modern compilers will provide __________ when they perform such unsafe conversions.
Floating-point less shocking:

```java
double googol = 1e100;
float tooSmall = googol;
```

Unless it's not:

```java
double pi = 3.1415926;
int piIsExactlyEqualTo = pi;
```

The same is true of floating-type types: a single-precision floating-point number (float) has fewer bits than a double-precision floating-point number (double), so it can’t represent numbers to the same precision (the clue's in the name!) or of the same size. A Googol can’t fit in a 32b float, so it’s not very surprising that copying a double into a float might end badly.

What's perhaps even more surprising is that the compiler will implicitly convert a floating-point number into an integer. This implicit conversion occurs through a process of *truncation*: 3.1415 is converted to 3, but so is 3.99.
Safe(ish)

```cpp
if, while, for
and, or, not
static_assert

int x = /* ... */;
if (x)
{
}

int bytes;
while (bytes = file.readSomeData())
{
}
```

The third kind of implicit conversion is the boolean conditional test. This is another information-losing conversion, but it is safe because it is not copying data into a new variable, it purely testing, "is this value non-zero?"
Summary of implicit conversions

Promotion

"Lossy" conversion

- Big to small
- Floating-point to integer

Boolean tests
Explicit conversions

"I know what I'm doing"

- static_cast
- dynamic_cast
- const_cast
- reinterpret_cast

In addition to implicit conversions inserted by the compiler, basically all programming languages allow the programmer to insert ________________ between types.

Explicit conversions are the programmer telling the compiler, "look, this might look unsafe to you, but trust me, ________________." Explicit conversions allow programmers to "break the rules" of the type system when they need to be broken — which shouldn’t be very often!

C++ has four built-in type casts, and C-style casts are also legal (though rather dangerous and therefore not recommended). We will look at all of these casts this term, starting with static_cast now and const_cast soon. The others will come up as we explain some prerequisite material.
The first C++ cast is **static_cast**. Its meaning will become much clearer when we talk about **dynamic_cast**, but for now, think of it has a _______________ : the programmer is endorsing the kind of thing that the compiler might do anyway. The compiler would be happy to convert a 64b integer to a 32b one without asking your permission, but a **static_cast** says that _______________.

This is the first C++ cast we’ll see, and you may notice _______________. It looks a little bit like a function call, but with a _______________ (a type name between angle brackets) between the cast name and the expression being casted. For example, in **static_cast<int>(some_long_value)**, the expression `some_long_value` is being converted to an int. _______________ explicit-conversions.cpp
Topic 4:

Variables and constants
\[ \nabla \times \vec{E} \]
\[ -\frac{\partial \vec{B}}{\partial t} \]
\[ \nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} \]
\[
\begin{bmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix} =
\begin{bmatrix}
l_{11} & 0 \\
l_{21} & l_{22}
\end{bmatrix} \begin{bmatrix}
u_{11} & u_{12} \\
0 & u_{22}
\end{bmatrix}
\]

A mathematical equation is simply a declaration that \underline{_________________________}. The left-hand and right-hand sides must be the same type of value, with the same units and the same value. The way that we use the equals sign in programming languages tends to be quite different.
bool b = true;
short s = 1000;
double x = 3.1415926;
string hi = "hello, world!";

More than just math:

- Name
- Space

Assignment in programming is about putting a value into a _____________. This is different from mathematical equality ($2x = y$), even though it uses the same equals sign. Instead, assignment is more like the definition of a mathematical variable (let $x = \frac{y}{2}$).

Just like mathematical variables, programming variables are names that can stand in for values. In this respect, double $x = 3.1415926$ isn’t so different from let $x = 3.1415926$.

Unlike mathematical variables, however, variables in programming languages are also associated with ______ ______.
Recall:

```
a = true
b = 42 > x
z = x + y
c = a or b
goto z if c
```

In our abstract machine model of a computer, variables take up _________________. The amount of space each variable requires, as well as the details of what its binary representation means and the operations we can perform with it, is _________________. 
Variables

Declaration:

```c
bool b;
short s;
double x;
```

Initialization:

```c
bool b = true;
short s = 1000;
double x = 3.1415926;
```

Space is set aside for variables by _____________ them. A declaration tells us the name of a variable and its type (which, in turn, tells us how much space we'll need to store it).

We often combine declaration with an initial assignment, or ________________. This is a very good idea, for reasons we'll see shortly.
Declaration:

```c
void doSomeWork()
{
    bool b;
    short s;
    double x;
    // ...
}
```

When we declare a variable but don’t initialize it, the compiler will set aside space for that variable in memory but not put anything in it. However, _______________ __________: every bit in the assigned space must always be either a 1 or a 0. So, if we don’t put a value into our newly-declared variable, it will contain _______________ __________. Sometimes they might happen to be all 0’s, but they could be any other value, including bits left over from some previous variable that we’re not using any more. Without initialization, we can’t know — more later on why this can be a problem.
Initialization:

```c
void doSomeWork()
{
    bool b = true;
    short s = 1000;
    double x = 3.1415926;
}
```

Variables

 memory

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0000 0001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>0000 0011 1110 1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>0100 0000 0000 1001 0010 0001 1111 1011 0100 1101 0001 0010 1101 1000 0100 1010</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initialization: the compiler sets aside space in memory for our variable, but it also fills that variable with a known value. The compiler emits instructions that will make our abstract machine copy the value on the right-hand side into the variable on the left-hand side.

Example: variables.cpp

Before we look at why this is so important, we need to take a more detailed look at these two sides of an assignment operation.
This two-line example of C++ code contains four expressions: `b`, `true`, `s` and `1000`. Actually, the assignments `b = true` and `s = 1000` are also expressions, but we won’t worry about that right at this moment! Looking at `b vs true` and `s vs 1000`, what is the difference between them?
What's the difference?

```plaintext
i = 42;
```

After the above assignment is performed, the left- and right-hand side assignment will have the same value: `i` will be equal to 1000. We could use these values almost interchangably: the expressions `i + 1` and `1000 + 1` will evaluate to exactly the same value. However, there is still an important difference between these

__________ ____________.
L- and R-values

L- vs... L???

```c
char c = 42;
int i = c;
```

Can only assign to an L-value

Can assign from any kind of value

Only L-values can appear on the left of an assignment, because only they have storage (space in memory) associated with them to put new values in. However, the right side of an assignment can be either an L-value (e.g., $x = y$) or an R-value (e.g., $x = 42$). The constraint on R-values is not that they’re the only things that can appear on the right of an assignment, it’s that

______________________________.
L- and R-values

```c
char c = 42;
int i = c;
double forces[] = { 4.1, 3.9, 2.4 };
```

L or R?

<table>
<thead>
<tr>
<th><code>c</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
</tr>
<tr>
<td><code>{ 4.1, 3.9, 2.4 }</code></td>
</tr>
</tbody>
</table>

On this slide, `c` is (perhaps unsurprisingly) an L-value and `42` is an R-value.

The third expression `{ 4.1, 3.9, 2.4 }` is an array initializer list: a list of values for the computer to put into the array. It's an example of a more complex literal value than a simple number, but it's still an R-value.Initializer lists like these have no names or storage associated with them: you cannot assign *into* them.

The fourth example, `forces[2]`, is a little more interesting again. `forces` is an L-value: it is the name for a place in memory where an array of three `doubles` are stored. However, `forces[2]` is __________: it is a name for a place in memory where a `double` can be stored. You can assign to it: `forces[2] = 0.0` is perfectly legal (and common!) code.
Previously:

```c
void foo(int x) { x = 42; }
void bar(int& x) { x = 42; }
```

More generally:

```c
int i;
int& j = i;
int& k = j;
```

Now that we’ve covered what an L-value is, we can try to make sense of a concept that may have been confusing in ENGI 1020: references.

You may have previously used pass-by-reference when calling functions. This means that a function’s parameter (in these examples, `x`) is not its own variable with its own storage: instead it ________ ______ ______ with a variable that was passed into the function.

This concept is more general than you may have previously seen: it can be used outside of function calls. In general, references are ____________________ ____.
References:

- are L-values
- refer to the same memory as another L-value
- referential transparency

```cpp
int i = 42;
int & j = i;
i++;
j++;
```

__________________________ They are names for storage in memory and they can be assigned to.

What’s special about a reference is that it’s an _________________. Another way of thinking about references is that _________________. For instance, in the real world, I have an office. I can refer to this physical location by many names: "my office", "EN3028", "the office across the hall from the computer lab", etc. I can ask someone, "please put that table in my office" or "please put that table in EN3028" and it has the same effect: ____________________.

This leads to the concept of referential transparency. This term means that doing something with or to a reference has exactly the _____________________. In the code example above, incrementing i has exactly the same effect as incrementing j.

_______________ references.cpp
L-values have memory

R-values do not

References are L-values that share
Assignment

```c
int i;
// ...
i = 42;
```

Variable assignment takes a value and stores it in an L-value. As seen previously, the left-hand side of an assignment must be an L-value: there must be a place to store the value being assigned.
Problems with variables

Uninitialized variables

```
int i;
// ...
if (i < 42)
```

```
uninitialized-variables.cpp:7:6: warning: variable 'i' is uninitialized when used here [-Wuninitialized]
if (i < 42)
  ^
uninitialized-variables.cpp:3:7: note: initialize the variable 'i' to silence this warning
int i;
  ^
1 warning generated.
```

Fortunately, we have some help to keep us from relying on uninitialized memory. When we compile software, we can tell the compiler to give us warnings about potentially unsafe constructs such as the use of uninitialized memory. With Clang and GCC, simply use the `-Wall` ("Warnings: all") flag at the command line (e.g., `clang++ -std=c++11 -Wall foo.cpp`).

Code example: `uninitialized-variables.cpp`
Problems with variables

More popular errors:

```java
if (i = 42)
{
    // ...
}
```

```java
while (someData[i]++)
{
    // ...
}
```

This slide shows another two popular errors. In the first case, it looks like we meant to check whether a value `i` is 42 or not, but in fact, we are assigning the value 42 to it. In fact, the assignment `i = 42` will evaluate to 42, which will evaluate to `true` because it is non-zero. Therefore, instead of executing some code if `i` is 42, we will set `i` to 42 and then __________ execute the code inside the "then" clause.

In the second case, it looks like we meant to iterate over an array: to look at each of its elements in turn then take some action until we reached the end (in this case, until `someData[i] == 0`). However, rather than incrementing the loop index `i`, we are actually keeping `i` the same and changing the value at position `i` within `someData`!
Here's what we meant:

```java
if (i == 42)
{
  // ...
}
```

```java
while (someData[i++])
{
  // ...
}
```

Oops!

These kinds of errors (_____________________________) can be prevented through the use of _constants_.


Remember preconditions and postconditions?

\[ \langle x \geq 0 \rangle x := x + 1 \langle x > 0 \rangle \]

Constants are contracts:

```cpp
const int x = 42;
```

I promise not to modify \( x \)

Constants are a form of contract, which is a concept we should have seen before in the form of preconditions and postconditions.

A precondition is a logical expression that some code ____________________________. The code may not execute correctly if its preconditions are not met. A postcondition is a statement of something will be true after the code runs _____________________________.

Together, the pre- and post-conditions provide a logical contract for some code: "if you give me some acceptable value of \( x \), I will return you some acceptable value of \( y \)."

Similarly, the use of the C++ keyword `const` is a contract, a promise from the ____________________________ to the ____________________________: I promise _____________________________.

A constant is not necessarily a guarantee that ____________________________. That is sometimes the case, but ____________________________.
If we had used the `const` keyword literally in the code above, we would have been protected from making these mistakes. We would not have been able to assign to `i` in the `if` condition, because we had promised not to modify `i`. We would not be able to modify the values in `someData`, because again, we had promised not to modify anything in `someData`.

Code example: `const-modification.cpp`
These promises also apply to references. A `const` reference to an L-value is a reference and a promise: you promise not to modify whatever memory you are referencing.

In order to keep this promise, any new references derived from a `const` reference must also be `const`.

It is an error to derive a non-`const` (read/write) reference from something that is `const`: the compiler ____________________________.
Another way that constants can be helpful is when we share data among multiple threads of execution. Concurrent programming is the topic of a Term 7 software course, but for now, it's enough to know that our computers can do increasing numbers of things at the same time, and that

In the example shown here, if thread1 and thread2 execute at the same time, it's impossible to predict whether the final value of $i$ will be 51 or 55. As programmers, this non-determinism is a bit of a problem: we don't want to write software whose behaviours ______________________. Put another way, we like ____________________.

Constants can help us write concurrent software correctly because ______________________ ____________________. If neither thread can modify $i$ then the value of $i$ becomes easy to determine again. If we need to combine the results of the two threads in some way, we can be clear and explicit about this --- using one of the many techniques that will be explained in Term 7!
Constants can actually make your software faster, too. In this code example, the programmer is using \texttt{const int DaysInWeek = 7; const int WeeksInSemester = 12; const int CourseDays = WeeksInSemester \times DaysInWeek;}. This is good for programmer understanding, and it’s also good for \underline{constants}. Your first instinct might be to write \texttt{i < 84} rather than use all of these named constants, but what if the University decides to change the number of weeks in a semester? If we have \underline{84} like 84 scattered around our code, it will be very difficult to answer the question, "which values depend on the number of weeks in a semester?" Using named constance, we can fix our code by modifying one value, \texttt{WeeksInSemester}. If we recompile our software, that new value will be picked up by all of our code without any manual intervention.

However, having to re-calculate \(7 \times 12\) every time we run the program is a little bit redundant. Fortunately, if the compiler can see that these values really are constant, then it will perform \textit{constant propagation}, effectively changing the loop to use the condition \(i < 84\) when it’s compiled. This gives the \underline{constant} with the \underline{84}. 
A simple loop:

```cpp
const std::string& s = getName();
while (x > s.length())
{
    // do something to x
}
```

This is another example of a loop whose performance can be improved through the use of a constant variable. In this example, we need to check the length of a string every time we go through the loop.

If, however, we know that the string isn't going to change length...
Constants as an optimization

A more efficient loop:

```cpp
const std::string& s = getName();
const size_t length = s.length();

while (x > length)
{
    // do something to x
}
```

... then we can pull out its length into a constant value, which we can check every time we go through the loop much faster than if we had to re-determine the string’s length.

Depending on the exact details of your code, the compiler may be able to _______________, but it's not guaranteed. Writing the code this way, however, it is.
In summary, then, constants can be useful for guarding against silly-but-easily-made mistakes (e.g., assigning to a variable instead of checking it for equality), difficult-to-do-correctly software patterns (e.g., shared mutable state) and can even help make our software faster.

For all of these reasons, when you create a new variable, it is good to ask yourself: "does this __________ to be mutable (i.e., non-\texttt{const})?" The answer is "no" more often than you’d think!
A common occurrence:

```c
void someBadOldAPI(string&);
int main(int argc, char *argv[])
{
    const string greeting = "Hello, world!";
    someBadOldAPI(greeting);
}
```

However, despite all of the advantages and uses of `const`, not everyone uses them all of the time. There may be times when you need to use an Application Programming Interface (API) that wasn't designed with thought to mutability and immutability (`const-ness`). For instance, in this code example, you must pass a non-`const` `string` reference into a function in order to, say, display it to the user. However, you only have a `const` reference, so what can you do?

In general, the only safe thing to do is to make a copy of the data into a new, non-`const` variable and pass the function a reference to that copy. However, if you know that the function isn’t actually going to change the data (e.g., in this case, it’s going to display it to the user and it could be marked `const` if the API authors had thought about it), then you can use a `const` to remedy the situation.
Explicit conversions

Recall:

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`

In our previous introduction to C++ casts, we saw some casts that weren’t elaborated on. One of these was `const_cast`.

All casts are a way of telling the compiler, "I know what I’m doing, it’s ok that I’m breaking the rules." In the cast of `const_cast`, the programmer is telling the compiler that, "yes, passing this value would break my promise demand `const` promises of others, but I’ve checked that it’s ok in this case."

Code example: `const-cast.cpp`
Can require:

- More care during design
- Less sloppiness during design!
- `const_cast`

The liberal use of `const`, then, can require a little more care in the design of software. Often this is because the use of `const` will expose sloppiness and danger in software design, so it’s well worth the extra work. When interfacing with APIs that take less care, however, `const_cast` may help you to ___ __________ without making the computer do lots of extra work ________________.
Explicit conversions

Variables

• L- vs R-values
• References

Constants
Topic 5

Functions
One model:

\[
\text{let } f(x) = 2x \\
\quad f(5) = 10 \\
\quad f(42) = 84 \\
\quad \Theta = \frac{1}{2} \pi \\
\quad f(\Theta) = \pi
\]

One model of a mathematical function is of something that processes inputs and produces an output. In this model, some values go in and a value comes out. The function is like ______________ 
__________________.
Another model:

\[ f(x) = \pi \]

\[ 2x = \pi \]

\[ x = \frac{\pi}{2} \]

Another model of a mathematical function involves substitution. In this model, we can replace \( f(x) \) with \( 2x \) in an equation, allowing us to solve for \( x \). The function doesn’t mean anything beyond something that can be substituted in.
Our functions will look more like ________________: values go into them and values will come out. The idea of solving for a function input given and output by rearranging its internals is rarely simply enough to make sense.
Functions

Calling functions

Finding functions

Implementing functions

- parameters and arguments
- recursion
We can use a function call as _______________ (assuming the function returns a value). In this code example, we will print out the values of three expressions:

"sin(0) is approx. " : a string

fastSine(0) : a value that is approximately equal to sin(0)

"\n" : another string

(__________ we will explain how this printing actually works with the << operator later in the term)

The call to fastSine works like _______________ : the value 0 goes in, and whatever value comes out is the evaluated value of the expression fastSine(0).

__________ if you run this code example exactly as written here, you will encounter an error: we have used the identifier fastSine without ______ ______ _______ _______ what that name means.
We typically find function *declarations* in ____________ that we reference from source files using the `#include` directive. `#include` acts almost like a ____________ operation: whatever content is found in *fastmath.h* (including files that it `#includes`) are ____________ ____________ #include "fastmath.h" ____________.
Declaring functions

fastmath.h:

```c
double fastSine(double radians);
```

Parameters:

```c
bool f(int x, double y, string s);
```

The same as:

```c
bool f(int, double, string);
```

A function declaration tells us three things: the function’s return type, its name and information about its parameters. A ________________ : "I promise that there is a function called fastSine that will return a double when it is passed a double." Once the compiler has seen a function declaration, it is able to process calls to that function.

You should note that the information about the parameters that really counts is their ___________ and their ___________. Each parameter’s ________________ : the compiler only really cares that "the first argument to the function should be a double", etc.
Default arguments:

```cpp
double fastSine(double radians = 0);
```

```cpp
double x = fastSine(0.0001);  // x should be 0.0001
double y = fastSine();         // no argument: defaults to 0
```

Only at the end:

```cpp
int f(int x = 1, int y);
```

```
default-args.cpp:17:22: error: missing default argument on parameter 'y'
int f(int x = 1, int y);
```

It is also possible to specify default values to _______________. This specifies that, if an argument is not passed to a particular parameter, the given value should be used instead.

When calling a function that has default arguments, we can _______________ from the call. This only works for parameters that have default arguments.

Default arguments can only be used _______________: once you use a default arguments, _______________ must use default arguments.
Find the functions

Compile this example:

```bash
$ clang++ -std=c++11 main.cpp
```

Undefined symbols for architecture x86_64:
"fastSine(double)", referenced from:
_main in main-843726.o

main.cpp:

```cpp
int main(int, char*[]);
```

=> wants to call `double fastSine(double)`

The code we’ve seen thus far will not compile by itself. So far, the compiler has seen a
______________ for `fastSine` ("I promise this function exists") and a call to that function,
but no ______________ for the function.

Without a function __________, the compiler will be unable to link your compiled code
into a complete program: there is a piece missing. This is the meaning of the undefined
reference or undefined symbol error message: the compiler was promised that something
would exist, but it doesn’t.

When the compiler compiles a single C++ source file, it keeps track of all the functions that are
defined in that file and all of the functions that are ______________ from that source file. To
create a complete program, you must supply a definition for every referenced function. We’ll see
more about this in Lab 2, but for now, that often means that you need to _______ ______
______________________, one that defines the missing function.
This is an example of a function definition from the previously-mentioned additional source file. Like its corresponding declaration, the function definition starts by describing the function's ____________________. Unlike the declaration, however, it doesn’t end with a semicolon: it instead has a ____________________, a block of statements that say what should happen when the function is called.

It is usually a good idea to #include the corresponding header file from a source file. This allows the compiler to see both the declaration and the definition of a function at the same time and check that _____________________. If they are not, it’s better to notice the difference early at compile time!
$ clang++ -std=c++11 main.cpp fastmath.cpp

**fastmath.h and main.cpp:**

```cpp
double fastSine(double radians);

int main(int, char*[]);
  => wants to call double fastSine(double)
```

**fastmath.cpp:**

```cpp
double fastSine(double) { /* ... */ }
```

If we compile both main.cpp (which references `fastSine`) and fastmath.cpp (which defines `fastSine`), the compiler is able to generate a complete program.

_________ what will the name of the output program be?

_________ since we didn’t specify a name (e.g., `-o myprogram`), the compiler will default to `a.out` (or `a.exe` on Windows).

Code examples: [fastmath.h](#) and [fastmath.cpp](#)
Implementing functions

Parameters:

```c
double fastSine(double radians)
{
    // The small angle approximation: sin(x) = x
    if (radians < SmallAngle)
        return radians;
    return sin(radians);
}
```

Like local variables, but initialized by arguments

**parameterize the function**

Inside of a function definition, we can use parameter values as inputs to whatever computation the function is supposed to perform. They are ____________: named storage with values inside them. They can even, if not **const**, be modified like local variables.

In fact, parameters are ____________ within a function that the programmer ____________. They have initial values, but those values come __________ passed to the function when it is called.

Parameters are so named because they **parameterize** a function: they allow its ____________ ____________. For instance, a **sin** function wouldn’t be very useful if it could only calculate the sine of one particular angle! Instead, using parameters, we don’t need to know the value of \( \theta \) when we write the `fastSine` function: we write a general-purpose function that will work for any value of \( \theta \) that is passed into it.
Arguments:

fastmath.h:

```c
double fastSine(double radians);
const double sinOfZero = fastSine(0);
const double Pi = 3.1415926;
const double sinOf90 = fastSine(0.5 * Pi);
```

```
let sinOfZero = (call fastSine with radians = 0)
let Pi = 3.1415926
let sinOf90 = (call fastSine with radians = (0.5 * Pi))
```

*Arguments* are the other side of the coin: they are the values that are passed into a function when it is called.

If parameters are local variables that don’t have to be initialized, arguments are their initializers. If a function has an integer parameter / takes an integer argument (these phrases are used interchangeably), the combination can be treated like an assignment statement (e.g., `int parameter = argument;`). Just like assignment, parameters have to be L-values but ________________.

The *pseudocode* on this slide is a rough translation of the C++ above it. It shows how we call the `fastSine` function twice, once with ________________ radians ________________ and once with it ________________ 0.5 * Pi ________________. We call the same function twice, but we expect ________________ each time. We haven’t changed the function or its parameters, only the ________________.
Calling functions

fastmath.h:

```c
double fastSine(double radians);
```

fastmath.cpp:

```c
double fastSine(double radians)
{
    // The small angle approximation: sin(x) = x
    if (radians < SmallAngle)
        return radians;
    return sin(radians);
}
```

The result of evaluating a function call is whatever expression is returned by the function. When a `return` statement is executed, we perform two steps:

1. **evaluate the expression** to the right of `return` and
2. **immediately leave** the function, returning that value.

Any statements that come after `return` are ignored: we exit the function immediately. In this example, if we passed the value $\pi/2$ into fastSine, we would check to see if it's less than the `SmallAngle` constant (defined in `fastmath.cpp`). If it were, we would immediately exit the function, return the value `radians`.

So, if `radians` is not less than `SmallAngle` (and $\pi/2$ shouldn't be!), we evaluate the expression `sin(radians)` and return that value. Of course, evaluating `sin(radians)` requires another function call! We take our `radians` parameter and pass it as an argument to `sin`. Whatever value is returned by `sin` is then the result of the expression `sin(radians)`, so we take that value and return it from fastSine.
When we combine these ideas (calls, arguments and parameters), we can draw a graph of function calls. This call stack shows ____________________________, and this very explicit version also includes _____________________________. It is called a stack because, once a new function call is added to the top, the only way to get back to a lower call is to remove the top one: to return from that function. This is analogous to a stack of plates, in which the only safe way to get at a plate in the middle of the stack is to take other plates off the top first. We will come back to this idea of a stack later in the course (and, in fact in the Term 4 Data Structures course).
Call stack:

```
sin: /* ... */
fastSine: return sin(radians)
main: y = fastSine(0.5)
```

Call stacks are typically represented a little more succinctly than on the previous slide, however: we don’t normally show argument values. This more common representation of a call stack can be found in many places, from documentation to debuggers.
An important concept in the use of functions is ____________. In a *recursive* function call, a function calls itself, which may result in another call to itself and another one, etc. The only difference among all of these calls is the ___________________________ rather than anything about the function itself. This may be a counter-intuitive way of writing software, but it is often a very useful way of solving problems.
Example: factorial

\[ 4! = 4 \times 3 \times 2 \times 1 = 24 \]

\[ F_n = n \times F_{n-1} \]

\[ F_0 = 1 \]

1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800 ...

One example of recursion being helpful is implementing factorials. Factorials are used a lot in number theory and probability to answer questions like, ”if we have four coloured balls in a bin and choose two of them, how many different combinations could we get?” The answer to these kinds of questions help us understand topics as diverse as winning the lottery and forging digitally-signed documents.

Recall that the factorial of \( n \) depends on the factorial of \( n - 1 \). For instance, the factorial of 4 can be evaluated as \( 4 \times 4 \times 2 \times 1 = 24 \).

We can write this more generally (and succinctly) as \( F_n = n \times F_{n-1} \), so long as we also include the base case \( F_0 = 1 \). Given these two equations, we can calculate the factorial of any non-negative integer.
Factorial function

**factorial.h:**

```c
/** Calculate the factorial of n (n times n-1 times n-2 times ...) */
long factorial(int n);
```

Writing the declaration for such a function in C++ is straightforward: we will write a function called `factorial` that takes an `int` index (n) and returns a `long` integer (since factorial values can get very large).
Factorial function?

```c
long factorial(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
    if (n == 2) return 2;
    if (n == 3) return 6;
    if (n == 4) return 24;
    /* ... */
}
```

A very straightforward definition of this function might attempt to ________________, but this approach can't work in the general case: we could never stop listing possible inputs and outputs. Or, to be more precise, on a modern notebook/desktop computer, we'd need to have listed all $2^{31}$ possible non-negative input values.
Factorial function?

```c
long factorial(int n)
{
    const values[] = { 1, 1, 2, 6, 24, 120, /* ... */
    return values[n];
}
```

A similarly naïve approach might be to ____________________________ and use the index to look up the correct value. This is more concise than the previous approach, but it suffers from the same fundamental limitation: it requires that we write in source code all of the possible factorial values we might ever need to compute.
Calling functions

Factorial function

\[ F_0 = 1 \]
\[ F_n = n \times F_{n-1} \]

factorial.cpp:

```cpp
long factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

If, however, we recall the two simple equations that can be used to describe the mathematical meaning of factorial, we can write a much simpler function to calculate these values. We can start by using the same ______________ and ______________ to write code that, if \( n \) is 0, will return 1, but otherwise, will multiply \( n \) by the factorial of \( n - 1 \) and return that value.

_______________ that the factorial function only call itself with an argument that will get us "closer" to the base case. If factorial(12) were to call factorial(12), then it would call factorial(12) and on and on forever. This is the recursive equivalent of an __________ ________.
We can recursively calculate the factorial of 3 by calling `factorial(3)`, which calls `factorial(2)`, which calls `factorial(1)`, which calls `factorial(0)`, which returns 1. Each of the calls on the stack then `return f(x)`, `return f(x)`, `return f(x)`, `return f(x)` respectively, unwinding the stack until we get back to the `main` function.

With a very simple idea and a very simple function, then, we can calculate the factorial of whatever non-negative integer we like (as long as it fits in an `int` and the result fits in a `long`). If we need to calculate a larger number, we must simply do the `return f(x)` a few `___________` times.

This technique can be used for solving many kinds of problems, including `______________`. If I want to know `_______________`, I can write a function to consider all of the currently-legal moves, see how many of them win the game for me, then `______________` `______________` `______________` to check how many of the non-winning possibilities can lead to winning moves.
Summary

Calling functions

Finding functions

Implementing functions
Is it better to ______________________ or ______________________?

It almost sounds non-sensical to ask the question! Well, the same principle is at work in software.

When we need to write software, the First Thing To Do is almost never "open a text editor".
J K Rowling didn’t begin work on Harry Potter by opening Word and typing. "Mr and Mrs Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal, thank you very much." In fact, an empty document or a blank piece of paper are often associated with ______________, the terrifying sense that you will never be able to even start what needs to be done (does that sound at all familiar?).
When we start with _________ rather than ____________, it’s easy to miss the forest for the trees: to focus narrowly on details rather than the big picture. This is how we end up doing things like “just add another if statement here” or “make one more while loop there” until we don’t recognize our own code.

Does *that* sound familiar?
Instead, before we ever start to write code, it is essential that we do some pre-work to make sure that we understand the fundamental problem and its solution. This overall solution is much more valuable than any individual piece of software! Having hit on the right solution, we could ___________________ and ___________________________ if we had to: the key thing is the solution itself rather than the code.

There is no One True Way to design. Design is a complex activity that requires both expertise and experience. However, I will suggest one rough workflow that I find works well.
One approach:

- understand the problem
  - break it into parts
  - solve it by hand
- choose representations
- design algorithms
- write code

Before designing anything, you first need to convince yourself that you understand the problem. As in almost any Engineering discipline, it can be helpful to divide and conquer: break a large problem into smaller, more manageable parts. You can also convince yourself that you understand the problem by running through a solution or two by hand. If you can work out a procedure for solving the problem by hand, you can write an algorithm for a computer to solve it.

Next, it’s helpful to choose appropriate representations for values in the problem. If you can figure out the ________________________________, transforming it into a useful form is a much easier task. At the moment, the representations in our toolbox are limited, but __________ ________ over the term (starting today!).

Next, you can figure out the algorithms you need to solve the problem. If you can break your solution down into a series of steps or a flowchart, ________________________________.

Finally, after doing all of this design work, translating your solution into actual code should be ___________________________! This is the only part of the process that depends on the specifics of any particular programming language.
Exercise:

As an exercise, think about a biosignals problem like an ECG (electrocardiogram) machine. This machine can tell us lots about the health of the human heart, but let’s consider one simple measurement: the heart rate.

Questions:

1. How can an ECG machine represent the changing values of this signal over time?

2. How can the machine identify peaks in the signal, if the actual maximum voltage is not consistent on every heartbeat?

3. Sketch out an algorithm to determine the heart rate (in beats/min) from some data represented in the form you described above.

Answer as many of these questions as you can, submitting the answers to a text file called ecg.txt in the examples folder of your Subversion directory.

You have all the tools that you need at your disposal to start answering these kinds of questions. It takes practice to develop your algorithmic thinking and design skills, but you can start today!
We’ve started to think about designing software using the abstractions that are already available to us, but working with primitive types, strings, arrays and functions can be quite limiting.

Much of the business of programming is _________________, and most of the abstractions that we’ll define in this course rely on user-defined types.
User-defined types

- Enumerations (inc. C++11)
- Typedefs
- Structures
Name enumerable things

- Sunday, Monday, Tuesday, ...
- January, February, March, ...
- Red light, yellow light, green light
- Primary, secondary, undergrad, grad

Something *enumerable* is something that you can count (which is quite the opposite of "innumerable"!). Many things (in the real world, not just software) can be exactly __________ __________ __________. For instance, there are only twelve possible values for "the current month", and "the current month" is only ever exactly one of them (it’s never both February and July).
Course registration:

```c
void registerForCourse(string course,
                        string studentName, unsigned int studentType);
```

Registering a student:

```c
registerForCourse("ENGI3891", "John Doe", 2);
```

Imagine you are writing software for the Registrar’s Office, strictly sticking to the types we’ve learned about so far. If you work on a function that needs to know if a student is an undergrad or graduate, you might represent this information using an integer value. In fact, just to be safe, you might use four integer values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Primary</td>
</tr>
<tr>
<td>1</td>
<td>Secondary</td>
</tr>
<tr>
<td>2</td>
<td>Undergraduate</td>
</tr>
<tr>
<td>3</td>
<td>Graduate</td>
</tr>
</tbody>
</table>

In that case, registering an undergraduate student named John Doe for this course might look like the code example above: you could pass the number 2 into the `registerForCourse` function to indicate that John Doe is an undergrad.
Problems?

```c
void registerForCourse(string course, string studentName, unsigned int studentType);
```

```c
registerForCourse("ENGI3891", "John Doe", 2);
```

However, there are problems with this approach. Using numbers like this without explanation is sometimes referred to as using "magic constants" because they're a bit mysterious and mystical. The problems with magic constants are:

- their meaning is non-obvious (hard to read)
- numbers can easily be passed into the wrong places (hard to use)
- they can be confused with other uses of numbers (hard to maintain)
Problems with magic constants

Non-obvious meaning

```
registerForCourse("ENGI3891", "John Doe", 2);
printCourseOutline("ENGI3891", 2);
reserveTextbook("Stroustrup", 2);
```

So therefore...

In the code examples above, it's not clear what each use of the number 2 means. In the first case, we might remember that we represent undergraduate students with the number 2 when calling registerForCourse, but what does it mean for printCourseOutline? Does the 2 still mean "undergraduate"? Does it mean version 2 of the outline? What about with reserveTextbook? It could mean that the book is for an undergraduate course, but it could also refer to the second edition of the text.

This lack of clarity means that the code is ________________, which also implies something else...
Problems with magic constants

Difficult to maintain

Where do we use 2?

```java
registerForCourse("ENGI3891", "John Doe", 2);
printCourseOutline("ENGI3891", 2);
reserveTextbook("Stroustrup", 2);

int x = 2;
int y = 2 * x;
```

Because the code on the previous page is difficult to read, it’s also difficult to _______________. Imagine that you need to change the way that different levels of student are numbered, perhaps to be more specific about first year vs declared majors. If we need to change every instance of a 3891 undergraduate student into a "declared major undergrad", we might need to change every such use of the number 2 into the number 3.

The problem with this is that we have used the number 2 in a lot of places, and it’s highly unclear when 2 means "undergraduate" vs when a 2 is just a 2. Which of these 2’s do we need to change? It’s almost impossible to tell.
Problems with magic constants

"Oops, not that integer!

```c
void startNewTerm(string name, int level)
{
    // ...
    registerForCourse("ENGI3891", "John Doe", level);
}
```

But:

```c
// Hooray, I finished Engineering One!
const string me = "John Doe";
const int term = 3;
startNewTerm(me, term);
```

A related problem is that the system can be ______________. For instance, in the first code example here, we have defined a `startNewTerm` function with a `level` parameter. The fact that we call `registerForCourse` might clue us in to the fact that we’re using the 2-means-undergraduate scheme, but it’s easy to miss.

It’s even easier to miss when we call the `startNewTerm` function. In that case, there’s no indication at all that we should pass in 2 for undergraduate: we have no reason to think we couldn’t pass in 3 for Term 3, but that has a completely different meaning! This could cause the software to misbehave in ways that are difficult to diagnose.
What about documentation?

Documentation can help spot errors, but:

*It's better to prevent mistakes than to catch them!*

**Can't compile error? Can't run it.**

One approach to help programmers use the `registerForCourse` and `startNewTerm` functions correctly is to write *documentation* for them. If it's written down somewhere that the `level` parameter of `startNewTerm` means a "2-for-undergraduate" sort of level, then the programmer might have a chance of not passing the wrong value in.

_________________, it would be much better if, instead of having to trust all programmers to read the documentation thoroughly, we __________________ to __________________

_________________.
Improvement:

```cpp
const unsigned int Primary = 0;
const unsigned int Secondary = 1;
const unsigned int Undergraduate = 2;
const unsigned int Graduate = 3;

void registerForCourse(string course, string studentName, unsigned int studentType);

registerForCourse("ENGI3891", "John Doe", Undergraduate);
```

One level of improvement would be to use named constants for these values. In this scenario, we keep using the same values and types, but at least we can use symbolic names instead of magic constants (e.g., `Undergraduate` instead of 2).
Improvement, but:

```c
const unsigned int Primary = 0;
const unsigned int Secondary = 1;
const unsigned int Undergraduate = 2;
const unsigned int Graduate = 3;

void registerForCourse(string course,
                        string studentName, unsigned int studentType);
```

Could still pass wrong integer:

```c
registerForCourse("ENGI3891", "John Doe", term);
```

Since these named constants are still just integers, it’s still very easy to use the wrong kind of integer. In this example, we can still pass 3 (for Term 3) to a function that thinks 3 means Graduate Student.
Better:

```cpp
enum class StudentType
{
    Primary,
    Secondary,
    Undergraduate,
    Graduate,
};

void registerForCourse(string course,
                        string studentName, StudentType studentType);
```

A much better solution to this problem is the *enumeration*, specifically the C++11 *strongly typed enumeration*. We can declare an enumerated type using the syntax on this slide. We can declare any variable, parameter, etc., to be of this new type, meaning that value is a kind of student, __________to the `registerForCourse` function. Once again, the compiler stops us from running certain kinds of erroneous code.

___________the keyword combination `enum class` ____________ have anything to do with C++ classes: it's just how we spell "strongly typed C++11 enumeration".
To use a strongly typed enumeration, we use values like `StudentType::Undergraduate` or `Month::January` instead of 2 or 1. Like the named-constants approach, we now have clearer names than when we used magic constants, but a name like `StudentType::Undergraduate` is even clearer than just `Undergraduate` (as we will see).
Allowed (and intended!):

```cpp
register("ENGI3891", "John Doe", StudentType::Undergraduate);
```

Compiler error:

```cpp
register("ENGI3891", "John Doe", CourseType::Undergraduate);
```

One benefit that strongly typed enumerations bring, with their explicit names like
`StudentType::Undergraduate`, is that we can re-use a name like `Undergraduate` in
different types but keep different meanings. For instance, an undergraduate course is not the same
thing as an undergraduate student (or an undergraduate major or an undergraduate thesis or ...).
Using strongly typed enumerations, we can keep the distinction clear.

However, this wasn’t always the case!
Before C++11, we didn’t have strongly typed enumerations, only old-fashioned C enumerations (recognizable by just the keyword `enum`, not `enum class`). These enumerations didn’t use long, fully-qualified names like `StudentType::Undergraduate`, but as a result, you couldn’t use both `StudentType::Undergraduate` and `CourseType::Undergraduate`: you just referred to `Undergraduate` and the compiler got confused as to which one you meant.

```
enum StudentType { Primary, /* ... */ };  
enum CourseType { Undergraduate, /* ... */ };  
StudentType johnDoe = StudentType::Undergraduate;
```

It’s a good idea to always use C++11 strongly typed enumerations when you write new code, but you may be exposed to the older style when working on real code in existing systems.
Enumerations summary

Symbolic names

- More readable (clearer)
- More maintainable

Integer constants

C++11 vs C++98
Typedefs

Why?

- to simplify names (readability)

```cpp
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
```

- to have one definition (maintainability)

One simple reason to use `typedef` is to simplify type names. For instance, the type `basic_string<char>` is a bit cumbersome to type, so the standard library defines `string` instead, using a `typedef`.

A more substantial reason to use `typedef`, however, is that it allows us to define a type now, use that type's name in our code, and have the freedom to change our mind about the type's definition later. For instance, if I write some software that represents temperatures using integers, I could define my own type with code like: `typedef int Temperature`. If I did this, it would make it much easier to change my mind down the road and use `double` to represent a temperature. This seems like a major change, but if I've used `typedef` effectively, I might only have to change one line of code: the `typedef` itself.
Group of L-values ("fields")

```cpp
struct Course {
    const Faculty faculty;
    const Department department;
    const unsigned int number;

    const string instructor;
    const unsigned int creditHours;
    const unsigned int labs;
};
```

A *structure* is a type that allows us to group values together when they are __________ but have __________________________. For instance, in the code example above, we group a number of named values together to describe useful information about a student.

Each of the L-values in this larger L-value is called a *field*. 
We can initialize an instance of a structure with a _______________________ of values inside curly braces (optionally stating the field names explicitly).

We can access a field within a structure instance using the ________________ (which we will define more precisely later). For instance, if we have a Course called engi3891, we can access the number of credit hours in the course by writing engi3891.creditHours.
Q: can we assign to a field?

```cpp
const Course course =
{
    Faculty::Engineering, Department::ECE, 3891,
    "Jonathan Anderson", .creditHours = 3, .labs = 4
};

course.faculty = Faculty::Science;
course.department = Department::Math;
course.number = 1000;
```

We can modify a field in a structure instance _________ if:

- the field is not declared `const`: otherwise, we promise not to change it
- the instance is not `const`: otherwise, we’ve promised not to change any of its fields

___________ structs.cpp
More sophisticated abstractions

Old:

```cpp
void registerForCourse(string course,
            string studentName, int studentType);
```

New:

```cpp
void registerForCourse(Student&, Course&);
```

Structures allow us to build up more complex and useful abstractions. Rather than writing complex function signatures that take many parameters to represent various elements of a Student and a Course, we can declare Student and Course structures that contain all of the relevant information. Then, if we add more detail to a Student, the only code that we need to change is the code that actually _________ the fields of Student.
Groups of L-values

- Names
- Storage
- Assignable (if not const!)

Building blocks
User-defined types

Enumerations

Typedefs

Structures

... and classes!
Classes are one of the most important ways that we have of defining new types. Much of this course revolves around classes.
A class is like a ______________. Like a struct, a class is a way of grouping related pieces of data together. Like a struct, a class defines __________ that can be accessed using the dot (.) operator. However, there are other properties of classes that make them much more useful than basic structs.

The single most important difference between structs and classes is that classes incorporate __________, not just __________. You can write a struct to hold information about a student, a book or a bank account, but you can define a class to include both the information and ____________________________.

Classes can also have sophisticated relationships among each other, reflecting how one class can be a special case of another, etc. However, that material comes later in the course.
Classes and objects

One definition, many instances

**Cars**: my car != your car != Steve's car

**string class**: my name != your name

**BankAccount class**:

my account != your account != President's account

A class defines the ___________ and ___________ of many objects. We can have many instances — many objects — of a class, and they will all have the same representation and operations, but they all have their own data. Just like the way that different variables can have the same type but different values (e.g., int x = 0; and int y = 42;), ___________

________________________________________________________________________.
Struct:

```c
//! Money comes in discrete cents.
typedef int Money;

//! A very simple representation of a bank account.
struct BankAccount
{
    string owner;
    Money currentBalance;
};
```

What operations can we perform?

This simple example shows a `struct` that can hold some basic information about a bank account. This `BankAccount` structure can be used to pass information about an account around as a ________________, but on its own, it can’t *do* anything.

_________ we are using `typedef int Money` to make it easier to re-define the `Money` type down the road. An integer number of cents is a pretty simplistic model for money: it doesn’t even allow us to express the idea of multiple currencies!

We’ve said that types define both the ______________ of data and the ______________ that can be performed with/on it. However, a `struct` by itself doesn’t really allow up to define new operations, only a representation. To add operations, we need the extra power of the ____________.
Classes: data + code

Class:

```cpp
#include <iostream>

using namespace std;

int main()
{
    // Money comes in discrete cents.
    typedef int Money;

    // A very simple representation of a bank account.
    class BankAccount
    {
    public:
        Money balance() const;
        void deposit(Money);
        void withdraw(Money);

    private:
        string owner;
        Money currentBalance;
    };
}
```

The declaration of a class looks very similar to the declaration of a structure. There are two important differences that we'll see today:

1. **visibility keywords** (here, public and private) and
2. **methods** (here, balance, deposit and withdraw).
Class definition is a statement:

```cpp
//! A very simple representation of a bank account.
class BankAccount
{
    /* ... */
};
```

Like a structure, the declaration of a class is a statement that specifies the _________ of the class, the ___________________________ between curly braces. Also, as a statement, it _________

Older compilers, when this semicolon was missing, would often emit confusing messages about the next thing they saw after the class declaration. Fortunately, modern compilers like Clang (the compiler we’re using in EN3000/3029 and the compiler that is included with Apple’s XCode) provide more helpful warnings like the one above.
Fields:

```cpp
//! A very simple representation of a bank account.
class BankAccount
{
    /* ... */
    string owner;
    Money currentBalance;
};
```

```cpp
BankAccount savingsAccount;
cout << savingsAccount.currentBalance;
```

**Exercise:** better representation for the account owner?

Like structures, classes define *fields*: variables that objects of the class will contain. Every object of the class is independent, with independent memory for its own fields: one `BankAccount` object may have one `owner` and `currentBalance`, another `BankAccount` object may have different values for these fields.

Accessing fields is done in the same way as for structures: with the dot (.) operator (which we will define more precisely in the [namespaces lecture](#)). In this slide’s second code example, we declare a `BankAccount` object: an object named `savingsAccount`. We then access `________ _________ currentBalance field with the expression savingsAccount.currentBalance.`
Visibility:

```cpp
//! A very simple representation of a bank account.
class BankAccount
{
    public:
    // ...

    private:
    // ...
};
```

The first major difference we'll see between structures and classes is the presence of visibility keywords. In this case, we have declared that certain members of the class are public, meaning that they can be accessed by any code. Other members are private, meaning that they can only be accessed by ________________ (this will make more sense after we see methods in a couple of slides).
Visibility:

```cpp
BankAccount savingsAccount;
cout << savingsAccount.currentBalance;
```

**error:** 'currentBalance' is a private member of 'BankAccount'

When we try to access a private member of a class (in the case of this code example, a private field), the compiler presents an error.
Visibility:

```cpp
// A very simple representation of a bank account.
class BankAccount
{
    public:
        // ...
    private:
        // ...
};
```

There is a third visibility keyword, `protected`, but its meaning won’t make any sense until we’ve covered more material later in the term.

_Why do we bother adding `private` members to a class if we can’t access them?_  

_They are still important parts of the ____________, and as we will see in a slide or two, we can access them _________________._
The second major difference between our structure and class versions of `BankAccount` is the presence of `methods`. Methods are _______________. They are declared just like regular functions inside of the `class` declaration, but they have the special meaning of being ________________.

Unlike a more traditional function, which acts only on its parameters (and on global variables — but we don’t want to use those!), a method is called _______________. For instance, in this slide’s second code example, we are calling the `deposit` method __________ `chequingAccount` __________. This means that we are depositing `someMoney` specifically ________________.
Class:

```cpp
//! A very simple representation of a bank account.
class BankAccount
{
    public:
        Money balance() const;
    /* ... */
};
```

Another special feature of methods is that the `const` keyword can be applied __________ ________, not just to its parameters or return type. When `const` is used in this way, it is __________: its fields will have the same values after the method call as they did before. In this example, the `balance` method promises not to change the `BankAccount`'s balance.
Class:

```cpp
//@ A very simple representation of a bank account.
class BankAccount
{
    public:
    std::string owner() const;
    Money balance() const;
    void deposit(Money amount);
    void withdraw(Money amount);

    private:
    std::string ownerName;
    Money currentBalance;
};
```

Like a `struct` with methods

A class, then, is an aggregation of fields (L-values within an object) and methods (functions that apply to an object) that can have visibility keywords applied to control who can access which members (fields and methods).

However, despite all of these details, it is perfectly reasonable to think of a class using one Big Idea:

-----------------------------------------------
A method implementation looks almost exactly the same as a function implementation. This shouldn’t be terribly surprising, as a method is almost exactly the same as a function. The key difference is that a method is a function ____________________.
Like a function:

- return type
- name
- parameter(s)
- body

Just like ordinary functions, every method definition has a return type, name, parameter(s) and body:

**Return type**

This is the type of value that the method will return. If the method doesn’t return a value, this should be `void`.

**Name**

A method has a name, just like a function. However, this name is a bit more complicated than just an identifier (see next slide).

**Parameter(s)**

Parameters for a method work exactly like functions.

**Body**

This is where the method is implemented. It’s very much like an ordinary function’s body, but it is also allowed to access the fields of an object.
Implementing methods

Money BankAccount::balance() const
{
    return currentBalance;
}

Different from a function:

- name is "inside" the class
- const method
- access to object fields

Name

Unlike a function, the _________________. We can say that the full name of the foo method in the Foo class is Foo::foo. Or, in this example, the full name of the BankAccount class' balance method is BankAccount::balance. We can see this if we compile BankAccount.cpp and ____________:

$ nm BankAccount.o | c++filt

Const

Unlike functions, methods can be marked with the const keyword. This means that the method _________________. That means that the method can read from fields of the object, but it _________________.

Object fields

A method is also allowed to access fields of its object, no matter if they're public or private. The method accesses them simply by name, _________________.

In course outline (3891/media/documents/outline.pdf):

4. Major topics

7. Classes

- define classes in terms of structures
- distinguish between classes and objects
- design, implement and call C++ methods
- explain how const can be applied to a method
Recall the `BankAccount` class that was introduced in lecture 8. In that lecture, we saw how a class can have fields (e.g., the current balance) and methods (e.g., `balance`, `deposit` and `withdraw`), but we didn’t answer one important question: how are objects of a class (particularly its fields) initialized?
Suppose that we declare two objects of class \texttt{BankAccount} (or, to use less specific language, two \texttt{values} of \texttt{type} \texttt{BankAccount}). In the declaration shown on this slide, we name one of these objects \texttt{chequing} and the other one \texttt{savings}. We then withdraw 500 units of money from the \texttt{savings} account to deposit in the \texttt{chequing} account.

We might then reasonably wonder, "how much money will we have in each of our accounts?" \texttt{savings} should have 500 units less than before and \texttt{chequing} should have 500 more, but how much does each contain \underline{\hspace{1cm}}?

The answer depends on \underline{\hspace{1cm}}, and as we have written the \texttt{BankAccount} class so far, \underline{\hspace{1cm}} \underline{\hspace{1cm}}.

\texttt{accounts.cpp}:

```
BankAccount chequing, savings;
chequing.deposit(savings.withdraw(500));
```
Every BankAccount object will have a currentBalance field of type Money (which, in BankAccount.h, is actually just an unsigned int). Therefore, every BankAccount object needs enough space set aside in memory for it to store an integer (32b = 4B on most computers today).

why do we only show the field as requiring memory in each object, excluding the methods from this picture?

the methods are the same for every object, so they don’t occupy space in each individual object.

When we declare a BankAccount object, then, we set __________ for it, but just like an uninitialized integer declaration (e.g., int x;), ________________ _________________. Therefore, the contents are undefined: the currentBalance could contain any value, depending on what that place in memory was previously used for.
What we want to do is to ensure that the `currentBalance` field is properly initialized to a reasonable value when a new `BankAccount` is created. For example, we might like to ensure that every `BankAccount` starts out empty (i.e., with zero units of money in it).
Recall:

```c
void doSomeWork()
{
    bool b;
    short s;
    double x;
}
```

This is the same problem that we observed with uninitialized variables in lecture 3. In that case, we ensured that variables were initialized using an *initializer*: a value that the variable would be set to from the very beginning of its scope.
Recall:

```c
void doSomeWork()
{
    bool b = true;
    short s = 1000;
    double x = 3.1415926;
}
```

This simple initialization technique worked for variables, so one might imagine doing something similar with objects...
Constructors

How about an initializer?
accounts.cpp:

```cpp
BankAccount chequing = {.currentBalance = 0};
BankAccount savings = {.currentBalance = 0};
```

No!

- `currentBalance` is private
- classes have another initialization mechanism

Could we use a simple initializer like we did with primitive values (integers, floating-point numbers, etc.) or with structures? If BankAccount were a structure, we could initialize it as shown in the example code on the slide.

This technique does not work with objects, however. Firstly, private fields can only be accessed by methods of a class, and we are not in such a method. More broadly, however, objects have a very different mechanism for initializing fields, partly ________________.
The purpose of classes is to ______________________, allowing programs to be written at higher levels of abstraction. Requiring explicit initialization of objects’ fields would violate this principle by requiring all code that creates objects to fully understand the internals of those objects.

In the example on this slide, the Coordinate2D class allows objects to be created that represent points in a two-dimensional space. These coordinates can then be accessed or compared using Cartesian co-ordinates (useful for drawing and other applications) or polar co-ordinates (useful for collision detection, gravity simulation and other applications). Internally, a Coordinate2D object might store its position with Cartesian, polar or some other co-ordinate system, but __________ __________ __________ Coordinate2D __________ about these implementation details.

So, then, how can we __________________________________________________________
_________________________________________?
Constructors initialize objects:

```cpp
class BankAccount {
    public:
        BankAccount();
    Money balance() const;
    void deposit(Money amount);
    Money withdraw(Money amount);
    private:
        Money currentBalance;
};
```

The answer is that we initialize an object’s fields with ________________ ____________. Since they are methods, constructors are declared inside classes. As shown on this slide, a constructor is distinguished from other methods by two characteristics:

- a constructor has the same name as its class and
- a constructor has no return type.

The constructor will be ________________. For instance, when we create a BankAccount object with the declaration BankAccount savings;, two things will happen:

- space will be set aside for the BankAccount object and
- the BankAccount::BankAccount() constructor will run.
Constructors initialize objects:

```cpp
BankAccount::BankAccount()
  : currentBalance(0)
{
}
```

```cpp
BankAccount::BankAccount()
{
  currentBalance = 0;
}
```

There are two ways, syntactically, of doing this. The first is with an *initializer list*, shown at the top of this slide. This version of the constructor shows:

- **no** return type (since it's a constructor),
- the method's full name (prefixed with `BankAccount::` like any other method of this class),
- a parameter list (like any other method), empty in this case,
- the *initializer list*, which initializes the `currentBalance` field with the value 0 and
- a method body, empty in this case.

The second version of this constructor looks a little more like other method that we've already seen: there is no initializer list, we instead assign the value 0 to the `currentBalance` field in the method body. This version of the constructor may look more familiar, but ______________ ________________, for reasons that are discussed later in this lecture.
Constructors can have parameters:

```cpp
class BankAccount
{
    public:
        BankAccount(std::string bankName);
    /* ... */
    private:
        const std::string bankName;
        Money currentBalance;
};
```

 Constructors, like other methods, can also have parameters. The constructor on this slide expects to receive a single argument of type `std::string` that represents the name of the bank. The argument to this parameter can be passed in parentheses after the object’s name, in braces or — because the constructor only takes one parameter — a single value on the right side of an initializing assignment operator.
Constructors

BankAccount.cpp:

```cpp
BankAccount::BankAccount(string bank)
    : bankName(bank), currentBalance(0)
{
}
```

BankAccount.cpp:

```cpp
BankAccount::BankAccount(string bank)
{
    bankName = bank;
    currentBalance = 0;
}
```

This example shows one of the shortcomings in using the familiar, just-assign-to-the-fields pattern for our constructors. On this slide, the first version of the constructor uses an initializer list to assign the value `bank` to the `bankName` field and the value `0` to the `currentBalance` field. This constructor compiles and runs as we’d expect it to.

The second version of the constructor attempts to assign the values `bank` and `0` to the `bankName` and `currentBalance` fields, respectively. This constructor fails to compile, however: the compiler complains that we can’t assign to the `bankName` field because it’s `const`. This is the essential difference between the initializer list and the constructor’s body: the initializer list is used to ________________________________, whereas assignments in the constructor body are ________________________________. In the case of `const` fields, such after-the-fact assignments are not allowed, because ________________________________

This is one reason why ________________________________.
Constructors

Constructor guidelines

- KISS
- Preserve invariants

This problem leads us to two guidelines for writing constructors:

1. **Keep It Simple**, $\{\text{Something that starts with } S}\}$, and
2. you must establish or preserve **object invariants**.
This slide shows an example of a potentially problematic constructor. The constructor seems innocent enough: it takes an account number passed into it as a parameter, uses it to look up details of a customer account in a database, retrieves the current balance and uses that balance to initialize its `currentBalance` field. This seems to be logical and straightforward, and indeed it may be fine __________________________.

However, very few programs will have the luxury of running on perfect platforms with perfect users. On practical systems, _______________ (more about this in lectures 14 and 15).

______________ what will happen if either of the errors described on the slide occurs during the construction of our `BankAccount` object?
Invisibly-invalid objects:

```java
BankAccount chequing(12345678);
chequing.withdraw(500);
```

When constructors fail, the result is _____________________________. For instance, if the previouslyShown constructor were passed an invalid account number, we would end up with a `BankAccount` object that does not represent a bank account, whose balance is unknown and whose interactions with other bank accounts would spread incorrect information.

This is because _____________________________. Error handling and exceptions are discussed in lectures 14 and 15, but for now, we'll say that the normal ways in which we might handle an error cannot apply to constructors.
Testing:

BankAccount chequing(12345678);  
chequing.withdraw(500);

Can only test `withdraw` method if connected to the real database!

Another problem with the previously-proposed constructor is that ______________________. Since the constructor performs a real database lookup, we cannot test our BankAccount class in isolation from the others.
Instead of:

```c++
BankAccount chequing(12345678);
chequing.withdraw(500);
```

Keep the constructor simple:

```c++
unsigned int accountNumber = 12345678;
AccountData info = database.lookupAccount(12345678);
if (info.valid())
{
    BankAccount chequing(accountNumber, info.balance());
    chequing.withdraw(500);
}
```

If we follow the KISS constructor pattern, all potentially-error-inducing operations are performed outside the constructor, in code that is able to report or handle errors. All the constructor does, in this case, is copy values into the object’s fields, which is not subject to the same kinds of potential errors. This means that, if we got as far as running the constructor by instantiating an object, we must have gotten past the errors that might’ve occurred. Alternatively put, ______________ ________________.
More testable objects:

```cpp
unsigned int fakeAccountNumber = 12345678;
AccountData info = { /* some test data */ };

BankAccount chequing(accountNumber, info.balance());
chequing.withdraw(500);

// now check that chequing.balance() is correct
```

The KISS pattern also _________________________________. After removing the
database-interfacing code from our constructor, we are left with a `BankAccount` class whose only
concern is managing a bank account. This functionality can be ____________________________ from
the rest of the program, enhancing our confidence in its correctness.
KISS:

```cpp
SomeClass::SomeClass(/* ... */)
    : field1(parameter1), field2(parameter2), // ...
{ }
```

Ask:

Can this constructor fail?

Ideally, constructors will look like the example shown on this slide: ________________
______________________________. It's not always possible or convenient to simplify
things this much, but it's the ideal that we strive towards.

Crucially, one should ask of one's constructors, ________________
__________ If the answer is "yes", consider redesigning the constructor and moving the potentially-
error-inducing code outside of the constructor.
The computer knows how to make a copy of a primitive value like an integer: it just needs to copy the bytes from one value to the other. The compiler’s job here is straightforward: it just needs to translate our source code into the CPU instructions that copy the values.
Contrived example:

```cpp
class Something {
    public:
    Something(double value);
    /* ... */

    private:
    double someValue;
    Time createdAt;  // when the object was constructed
};
```

When copying objects, things are more complicated. Even in simple cases, we need to _________  
______________________ from the original (source) object into the new (destination) object.  
Sometimes, however, things are more complicated than that.  

In the example shown above, we have two fields. One is a double-precision floating-pointer number:  
the sort of primitive value that the computer knows how to copy. The second field is more interesting. In objects of class Something, we have an _______________ that says the  
createdAt field should always contain _____________________________. If we  
were to simply copy the value of this field from an old object into a new one, _______________  
_____________________________. This would violate our invariant.  

Instead, what we want to do is to _______________________________. This would  
allow us to copy the fields that can be straightforwardly copied and do something special with the others.
The tool that we use to specify this copying behaviour is the *copy constructor*. In this example, the copy constructor for the `Something` class is `Something::Something(const Something&)`. This constructor will take a reference to another `Something` object and use it to initialize the `Something` object that’s under construction.

If our class only contains fields that are primitive values or objects, we don’t need to write a copy constructor, and ___________________________. This compiler-generated copy constructor will simply copy each of the fields from the original object to the new one, copying primitive types’ bytes and invoking objects' own copy constructors.
Copy constructors

Same name as class:

(like regular constructor)

One parameter:

reference to object of same class

Like every constructor, the copy constructor’s ______________________________. What distinguishes the copy constructor from other constructors is the parameter that it takes:

const _________________________________. This is the object that we will copy values from.
Returning to our contrived example, this slide shows both of the `Something` constructors: the "normal" constructor (which takes a `double`) and the copy constructor (which takes a `const Something&`). Both constructors are using `:` to initialize their fields, but the copy constructor gets its `someValue` value from the original `Something` object (called `other` in this case).
Copy constructors

What if we don't want copying?

```cpp
BankAccount savings;
savings.deposit(500);

BankAccount scammer(savings);
scammer.withdraw(500);
```

In some cases, however, we don’t want to be able to copy an object. For instance, an object that represents a resource like money, open files or dynamically-allocated memory (see lecture 18) might need to prevent copying in order to maintain its invariants (e.g., memory ownership, as in lecture 21). For such classes, we don’t want to write a copy constructor, but neither do we want the compiler to generate one for us.
What if we don't want copying?

```cpp
class BankAccount
{
    public:
    BankAccount(const BankAccount&) = delete;
};
```

Only in C++11 and up!

When we want to ensure that a class has no copy constructor (as in this case, where copying a `BankAccount` could mean copying money), we can explicitly tell the compiler not to generate one for us by declaring the copy constructor to be `_____________` (= delete).

This syntax only works in C++11 and up, however. When using C++98, software authors tend to declare the copy constructor but purposefully not implement it, leading to an `_____________` `_____________` when linking the program if anyone tries to use the copy constructor.
Copy constructors

With deleted copy constructor:

```cpp
BankAccount savings;
savings.deposit(500);
BankAccount scammer(savings);
scammer.withdraw(500);
```

accounts.cpp:19:14: error: call to deleted constructor of 'BankAccount'
BankAccount scammer(savings);
^       ~~~~~~~
./BankAccount.h:9:2: note: 'BankAccount' has been explicitly marked deleted here
BankAccount(const BankAccount&) = delete;
^ 1 error generated.
```

If you `accounts.cpp`, you can see the effect of trying to copy an uncopiable object. Using the C++98 technique we would see an undefined reference to the `BankAccount` copy constructor, which might not make your intention very clear. Using the `C++11 = delete` technique, however, the compiler can tell us that we are attempting to call a deleted copy constructor: it’s much clearer that `BankAccount`.
```
Copy constructors

What shouldn't we copy?

External resources

• money

• open files

Large data structures

In general, copying objects that represent external resources (e.g., money or files) can violate the logic of our program, causing incorrectness. For instance, a bank ledger program that duplicated money could not provide a very accurate accounting!

In many cases, we also don't want to copy very large data structures. This may not be for reasons of logical correctness but for efficiency. For instance, there is nothing incorrect about passing a large matrix object by value as a parameter, but it may waste much more time and memory than simply passing a reference.

There are other kinds of special resources that we don't want to copy, the most common of which (pointers to dynamically-allocated memory) is discussed in lecture 18 and lecture 21.
Summary

Constructors

Copying

• pass-by-value

• copy constructors
An important idea when we think about classes, objects, methods and calls is the distinction between compile- and run-time artifacts. For example, when declaring a simple variable like `int x`, everything that we need to know about the type `int` is known at compile time, whereas the key information about `x`, 

This distinction also comes into play with classes and objects. We can call `x` above an instance of the `int` type or `PI` and instance of the `double` type, and similarly we can call `savings` and `chequing` instances of the `BankAccount` type. However, there is also a more specific term: 

__________________________.
We also sometimes refer to things that can be known at compile time as ____________; things that might depend on values in memory at run time are ____________.

"Hello, world!": **static**

- in the standard "hello world" program, "Hello, world!" is a literal value typed in the code: **changing it would require re-compiling the source code**.

"I am compiled for Windows": **static**

- although this string might not be included in the Linux version of a program, it is still static: it depends on **whether we are compiling for Windows**, not any value held in memory at run time.

"I am running on LABNET-42": **dynamic**

- this string can’t be known at compile time: it depends on which computer actually runs the program (which is **dynamic, run-time** information).

"The time is 3:08pm": **dynamic**

- this depends on the current time **at run time**: it can even change during the program’s execution as the time changes.
Method names

```cpp
cout << "The balance is: " << savings.balance() << endl;

Money BankAccount::balance() const
{
    return currentBalance;
}
```

This static vs dynamic distinction also comes up in the way that we name methods. Methods run
"run the balance method on a particular BankAccount object". In this case, we are asking,"what is the balance of the savings account?"

When we are implementing a method in code for the compiler, however, we don't know in advance
what all of the objects will be: we define methods ________________. Because of this, we
don't use a dynamic name like savings.balance, we use a ________________ like
BankAccount::balance.
These two kinds of names use two kinds of operators. Static names like `BankAccount::balance` (or `std::string`, as we'll see later) use the `::` (scope resolution) to separate name components. The names that are separated by the scope resolution operator must be static names: `BankAccount::balance`.

Dynamic names are separated by the `.` (structure reference) operator. The name on the left of this operator is a dynamic name: it's the name of an object or a structure instance, a `savingsAccount` with a `balance` to hold dynamic values.

Comparison with other languages

Some programming languages do not have this distinction between scope resolution and structure reference operators. For instance, Java and Python use a dot (`.`) for both meanings. Other new languages like Rust maintain the static/dynamic naming distinction. No matter what programming languages you end up using, it's worth understanding the difference (even if your language doesn't make it explicit).
The idea of static naming is further extended in C++ through the idea of static
______________.
Have you wondered:

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

int main(int argc, char *argv[])
{
    cout << "Hello, world!\n";
    std::cout << "Hello, world!\n";
    return 0;
}
```

What's the difference?

Revisiting the classic "hello, world" program (possibly the first program you've ever seen), you may have wondered:

1. what does `using namespace std` mean,
2. where does `cout` come from and/or
3. what is the difference between `cout` and `std::cout`?

One goal of this course is the _________________: code that was previously a kind of magic incantation should become clear, so that you can _________________ more __________ by the end of this term. By the end of this topic, you should understand what `namespace std` and all of the related code means.
Short names are ambiguous

- 3rd floor
- 95

Fully-qualified names

- 3rd floor, S J Carew Building
- 95 Water St, St. John's, NL, Canada

Short names are convenient but ambiguous. The name "the 3rd floor" could refer to the 3rd floor of the S J Carew building in the context of a lecture, but it could mean the 3rd floor of some office building in the context of a job interview. The name "95" could be an office number, a street number, an apartment number, a PO Box number, etc. The name "Raymond’s" means very different things if talking to friends (in which case "Raymond’s" might be a friend’s house) or the organizing committee of a successful conference (in which case "Raymond’s" might mean the excellent restaurant on Water Street).

Fully-qualified names remove ambiguity but are less convenient to use. These unambiguous names remove confusion and are sometimes required, but it can be overkill to say, "the lab is on the 3rd floor of the S J Carew Building" when we are already in the S J Carew building.
Of course, if we want to use a __________ fully-qualified name, we should, strictly speaking, use the following address for Raymond’s:

95 Water Street
St. John's, NL
Canada
Sol 3
Milky Way
Local Group
Laniakea Supercluster

(assuming that everyone in the universe agrees on the name of our sun, galaxy, group, etc.!)
Namespaces

Naming problems

Collision:

```cpp
#include "SomeHeader.h"  // includes <string>
class string
{
};
```

Safe but ugly:

```cpp
class StringAsWrittenByJonathanAndersonAtMemorial
{
};
```

We encounter similar problems in programming languages. Suppose I want to define a class called `string`. There’s no reason that I can’t do this, so I do, and my code seems to work fine. If, however, I then add an `#include` as in the above code example, I could start to see a problem if the header I `#include` itself `#includes` a header that defines a `string` class. In that case, the compiler would see two different definitions of the name `string`: there would be a ________________.

One strategy for avoiding collisions would be to use names with a high probability of uniqueness. In the "safe but ugly" example, my new class name is unlikely to collide with any other class definitions, but it’s a rather unwieldy name to use.
C++ namespaces are designed to address this problem. Other programming languages use the same basic idea, but with different details and names. For instance, Java has packages, Python has modules, Rust has both modules and a high-level abstraction called "crates", etc. All of these techniques attempt to solve the same underlying problem: ____________________________

The goal is to use names that are neither over-qualified ("the EngCaf on the 2nd floor of the S J Carew building at Memorial University") nor under-qualified ("number 10"). Instead, we want to use ____________________________ . For instance, in the political press, the phrases "Number 10" and "White House" don’t need any further qualification to identify the residences of powerful people. When giving directions, however, some context might be needed ("Number 10, Elizabeth Avenue" or "the white house at the end of the street").
Namespaces provide a mechanism for using a "just right" (Goldilocks) amount of qualification for names. Namespaces are named places to define other names. We can use namespaces to define names for types, functions and values, all within the same namespace, which itself has a name. For example, we can define our own string class (as above) inside a namespace like engi3891.
In this example, we distinguish between the standard library’s `string` class (`string` within the `std` namespace) and our own (`string` within the `engi3891` namespace). The same applies for functions and named values like global variables: we can define new names in our namespace without fear of conflicting with things defined in other namespaces.

Just like there can be a Water Street in both St. John’s and Bay Roberts, there can be different `string` classes defined in multiple namespaces.
The C++ standard library defines its types, functions and values inside of the `std` namespace. This is a simple, well-known place to find things in the standard library, and only things in the standard library. That is: unless you become a maintainer of the C++ standard library, ___________  
_________________ `std` ___________.
Since namespaces are, themselves, things with names, we can put them inside of other namespaces.

If we want to be really, really clear about the name of our `string` class, we can put it in the `lecture9` namespace, which is inside the `examples` namespace, which is inside the `engi3891` namespace. Then, `engi3891::examples::lecture9::string` wouldn't even conflict with a class definition like `engi4891::examples::lecture9::string`. 
Can be as specific as necessary:

```
std::string foo = /* ... */;
engi3891::string bar = /* ... */;
engi3891::examples::lecture9::string baz = /* ... */;
```

When we use names that are defined in a namespace, we can use the complete (or __________ name) if we need to, making use of the scope resolution operator. When we want to be sure that we avoid any confusion ("which string did you mean?")", the fully-qualified name is unambiguous.

This is a little bit like referring to "Water Street in St. John's" or "Water Street in Bay Roberts", just to make sure there's no confusion.
Use short versions of long names:

```cpp
using std::string;
void foo()
{
   string bar = "Hello, world!";
}
```

"I will refer to 'std::string' as just 'string'"

In many situations, a short, potentially ambiguous name is actually fine in practice. Unless I know that you have a particular connection to Bay Roberts, I can just refer to "Water Street" and expect that you will know which one I mean. Similarly, when I use the class name `string`, I am almost always referring to the `string` class defined in the `std` namespace.

You can direct the compiler to use a short name for a namespaced name with the `using` directive. In the example above, `using std::string` means, "from now on, I want to be above to refer to `std::string` as just `string`.

The same technique works for names within nested namespaces:

```cpp
using engi3891::examples;
using engi3891::examples::lecture9::string;
/* ... */

string foo; // possible because of long 'using' directive
examples::lecture9::string bar; // possible because of short 'using' directive
```
Using many short names:

```cpp
using std::cerr;
using std::cout;
using std::iostream;
using std::string;
using std::vector;

void foo()
{
    string bar = "Hello, world!";
    cout << bar;
}
```

If we want to use many short names from a namespace, one option is to employ multiple `using` directives. However, this can get tedious (harder to write) and repetitive (harder to read).
Namespaces

Use all short names:

```cpp
#include "rates.h"
using namespace std;

void foo()
{
  string bar = "Hello, world!";
}
```

Not in header files!

A shorthand for multiple `using directives is the `using namespace directive. This allows ______________ to be made available within a source file. This is often used with the `std namespace to make `std::string, `std::vector, etc., available as just `string, `vector, etc.

This directive (or, in most cases, the vanilla `using) ________________________.
Topic 9:

Containers
Containers

Where do we store things?

Where do we store complex things?

- things with many parts?
- many things?

Our ________________ stores things in memory. The abstraction provided by a ________________ is the ________________. This is where we store things.

This is true whether a "thing" is something very simple (like an integer) or something more complicated.

When we want to store a thing with many parts, we use ________________. Both structures and classes define fields within a type so that one thing (e.g., a Student) can contain many things (e.g., a name, a student ID, an emergency contact number...).

When we want to store ____________________________, we use another kind of type: a ________________.
Previously: arrays

- multiple things in memory
- contiguous
- fixed size

We have previously stored multiple things in memory with __________. Arrays can store multiple L-values within a single (larger) L-value as long as each element has the __________.

Arrays are contiguous in memory: the first element is immediately next to the second element, which is immediately next to the third element, etc.

One limitation of arrays is that they have a __________. Once an array has been allocated to hold ten elements, it can never hold eleven. This is fine for some applications, but not all.
For example, a class that represents an intersection with traffic lights might hold four lights in four fields, each representing one traffic light on one side of the intersection. Using fields for these lights means that:

1. we are committed to a fixed number of traffic lights and
2. when performing operations on all of the lights, specific code must be written using the field names.
Containers

```cpp
class TrafficLight { /* ... */};
class Intersection {
    /* ... */
    private:
    TrafficLight lights[4];
};
```

An array-based alternative would store four lights in an array of `TrafficLight` objects. Now we can iterate over all of the lights more easily, but we still have the problem of a fixed size. What about intersections with only one light (e.g., Ropewalk Lane)? What about intersections with more than four lights (e.g., Kings Bridge Road)?
Containers

```cpp
class TrafficLight { /* ... */};

class Intersection
{
    /* ... */

    private:
    TrafficLight lights[5];
    unsigned int lightCount;
};
```

One option would be to allocate space for up to five traffic lights, together with a field indicating how many lights are actually held in the array. However, even this doesn’t cover spectacularly complicated intersections like the Magic Roundabout. If we made our array big enough to handle the world’s most complicated intersection, we would waste a lot of space when most intersections only have 2–4 lights. More importantly, we would need to change our code whenever a more spectacally complicated intersection got built somewhere!
Containers

Problems with arrays

Fixed size

Not always a good fit

Arrays, then, can be problematic because of their fixed, inflexible sizing. However, they are also _____________ for the requirements of every representation problem. For instance, sometimes we might need to _____________ of a list of elements. Arrays make this very difficult: we need to create a new, larger array, copy the beginning of the old array into the new array, copy the end of the old array into location $i+1$ of the new array, and finally assign the new element to location $i$. Similarly, it can be difficult to ____________

Because of these limitations, it is useful for us to think about a more general representation for storing "like" things. This abstraction is the container.
Containers

Things to put things in

Examples

- moving box
- std::list
- std::queue
- std::vector

In general (in real life as well as in programming), a container is simply a thing to put another thing in. A moving box is a container that can store books, clothes, dishes, etc. When designing data representations, a container is_________________________.

There are many types of containers with different properties, making them appropriate for different tasks. In ENGI 4892, you are likely to look at std::list, std::queue, std::map and others, but in this course we focus on std::vector.
Class in std namespace

Stores L-values of a type

- a vector of integers
- a vector of BankAccount objects
- a vector of GameBoard structures

`std::vector` is a class defined as part of the C++ standard library, so it is found in the `std` namespace.

Fun fact: (not required for this course) In reality, `std::vector` isn't actually a class, it's a template for a class. This is how we end up with `std::vector<int>`, `std::vector<std::string>`, etc. Using and writing templates is a very significant part of ENGI 4892.

Like all containers, `std::vector` stores L-values __________________. We can have a vector of integers (`std::vector<int>`), a vector of BankAccount objects (`std::vector<BankAccount>`) or a vector of GameBoard structures (`std::vector<GameBoard>`), but __________________________ __________________________.
Like an array

- holds elements of same type
- contiguous

Better than an array

- knows how big it is
- can grow to hold more things

In some ways, a vector is like a plain old array: it holds a number of elements of a particular type, and it holds them contiguously (next to each other). (some store things non-contiguously), but vectors do.

Vectors are also different from (and, in many ways, superior to) plain arrays. For one thing, a vector: if you have any kind of vector object, you can call its size method to ask how many elements it holds. This is better than having to pass around both an array and its size as independent values, since the two can get out of sync quite easily. With a vector, you will always know that the size is correct.

The second key difference from an array is that . If you create a vector with four elements in it but later discover that you need five, the vector can resize itself automatically. The internal mechanism for this isn’t simple (see assignment 6), but looking at a vector from the outside, it’s automatic.
Usage

Header file:

```cpp
#include <vector>
```

Declaration:

```cpp
std::vector<int> numbers;
std::vector<BankAccount> accounts;
std::vector<Student> students;
```

Using a vector is quite straightforward. First, you must `#include` the standard library header file it’s defined in (`vector`).

To create a `vector` object, simply declare it like any other variable: first the `___________` and then the `__________`. The examples from this slide show three vectors: one vector of integers called `numbers`, one vector of `BankAccount` objects called `accounts` and one vector of `Student` objects called `students`. 
Type parameter:

```
vector<int>
```

All elements of same type!

The type name inside the angle brackets is called a *type parameter*. The example declares that this particular vector contains integers, not bank accounts, students or any other type. 

```

```
Accessors:

```cpp
vector<int> numbers = /* ... */;

// C++98-style iteration:
for (size_t i = 0; i < numbers.size(); i++)
{
    cout << numbers[i] << "\n";
}
```

C++11 range-based iteration:

```cpp
for (int value : numbers)
{
    cout << value << "\n";
}
```

The next few slides show all of the methods of `std::vector` that you will be expected to remember. The `size` method can be used to determine how many elements the vector is currently holding. It returns a value of type `size_t`, which is an unsigned integer type (created with a `typedef` in the standard library header `cstddef`).

We can also iterate over a vector using the new C++11 `range-based for loop`. This style of iteration saves a little bit of syntax, but more importantly, it puts the focus of the code on the __________ ______________ rather than the mechanics of ______________. The implementation of range-based for loops is discussed further in lecture 19.
**Mutators:**

```cpp
vector<int> numbers = /* ... */;
numbers.push_back(42);

for (size_t i = 0; i < numbers.size(); i++)
{
    numbers[i] += 1;
}

numbers.resize(10);
numbers[8] = 8;
```

The `push_back` method adds an additional element to a vector, increasing its size by one. This is one of the primary advantages of vectors over arrays: resizing can be automatic. When desired, the vector can also be manually resized using the `resize` method: the example above resizes the vector to hold 10 elements.

`std::vector` also implements the index operator. We will discuss how this is done in lecture 13, but from a consumer perspective, we can use square brackets to access individual elements of the vector just like we would with an array. If a vector contains 10 elements, accessing element 10, 11, 12, etc., is a programmer error and could lead to unexpected results (e.g., a segmentation fault that terminates the program or, worse, modifying memory that doesn’t belong to you).
C++11:

```cpp
using std::vector;

void foo()
{
    vector<int> numbers = { 1, 2, 3, 4, 5 };
}

vector<Student> students =
{
    { .name = "Alice", .age = 19 },
    { "Bob", 20 },
};
```

In C++11 and later, vector initialization can look a lot like array initialization: a comma-separated list of values within curly braces.

The standards committee that developed C++11 put significant effort into improving the consistency of initializing containers, structures and even classes. __________________________
______________________________
C++98:

```cpp
using std::vector;
void foo()
{
    vector<int> numbers;
    numbers.push_back(1);
    numbers.push_back(2);
    numbers.push_back(3);
    numbers.push_back(4);
    numbers.push_back(5);
}
```

Earlier versions of C++ (e.g., C++98) did not provide a convenient mechanism for initializing vectors. Instead, it was common to either push_back (as shown in this slide) or else to resize.
Containers

- places to store multiple things

`std::vector`

- standard library container
- iteration
Topic 9:

Containers
Containers

Where do we store things?

Where do we store complex things?

- things with many parts?
- many things?

Our __________________ stores things in memory. The abstraction provided by a __________________ is the ____________. This is where we store things.

This is true whether a "thing" is something very simple (like an integer) or something more complicated.

When we want to store a thing with many parts, we use __________________. Both structures and classes define fields within a type so that one thing (e.g., a Student) can contain many things (e.g., a name, a student ID, an emergency contact number...).

When we want to store __________________, we use another kind of type: a ______________.
Previously: arrays

- multiple things in memory
- contiguous
- fixed size

We have previously stored multiple things in memory with _________. Arrays can store multiple L-values within a single (larger) L-value as long as each element has the ________ _________.

Arrays are contiguous in memory: the first element is immediately next to the second element, which is immediately next to the third element, etc.

One limitation of arrays is that they have a _______________. Once an array has been allocated to hold ten elements, it can never hold eleven. This is fine for some applications, but not all.
For example, a class that represents an intersection with traffic lights might hold four lights in four fields, each representing one traffic light on one side of the intersection. Using fields for these lights means that:

1. we are committed to a fixed number of traffic lights and
2. when performing operations on all of the lights, specific code must be written using the field names.
An array-based alternative would store four lights in an array of `TrafficLight` objects. Now we can iterate over all of the lights more easily, but we still have the problem of a fixed size. What about intersections with only one light (e.g., Ropewalk Lane)? What about intersections with more than four lights (e.g., Kings Bridge Road)?
One option would be to allocate space for up to five traffic lights, together with a field indicating how many lights are actually held in the array. However, even this doesn’t cover spectacularly complicated intersections like the Magic Roundabout. If we made our array big enough to handle the world’s most complicated intersection, we would waste a lot of space when most intersections only have 2–4 lights. More importantly, we would need to change our code whenever a more spectacularly complicated intersection got built somewhere!
Arrays, then, can be problematic because of their fixed, inflexible sizing. However, they are also ________________ for the requirements of every representation problem. For instance, sometimes we might need to ________________ of a list of elements. Arrays make this very difficult: we need to create a new, larger array, copy the beginning of the old array into the new array, copy the end of the old array into location $i+1$ of the new array, and finally assign the new element to location $i$. Similarly, it can be difficult to ________________

Because of these limitations, it is useful for us to think about a more general representation for storing "like" things. This abstraction is the container.
Containers

Things to put things in

Examples

- moving box
- `std::list`
- `std::queue`
- `std::vector`

In general (in real life as well as in programming), a container is simply a thing to put another thing in. A moving box is a container that can store books, clothes, dishes, etc. When designing data representations, a container is a data structure for storing and managing a collection of elements.

There are many types of containers with different properties, making them appropriate for different tasks. In ENGI 4892, you are likely to look at `std::list`, `std::queue`, `std::map` and others, but in this course we focus on `std::vector`. 
**std::vector**

Class in std namespace

Stores L-values of a type

- a vector of integers
- a vector of BankAccount objects
- a vector of GameBoard structures

*std::vector* is a class defined as part of the C++ standard library, so it is found in the *std* namespace.

*Fun fact: (not required for this course) In reality, *std::vector* isn't actually a class, it's a template for a class. This is how we end up with *std::vector<int>*,
*std::vector<std::string>*, etc. Using and writing templates is a very significant part of ENGI 4892.*

Like all containers, *std::vector* stores L-values _________________. We can have a vector of integers (*std::vector<int>*), a vector of BankAccount objects (*std::vector<BankAccount>*), or a vector of GameBoard structures (*std::vector<GameBoard>*), but ________________ ________________ ________________.
Like an array

- holds elements of same type
- contiguous

Better than an array

- knows how big it is
- can grow to hold more things

In some ways, a vector is like a plain old array: it holds a number of elements of a particular type, and it holds them *contiguously* (next to each other). (some store things non-contiguously), but vectors do.

Vectors are also different from (and, in many ways, superior to) plain arrays. For one thing, a vector : if you have any kind of vector object, you can call its `size` method to ask how many elements it holds. This is better than having to pass around both an array and its size as independent values, since the two can get out of sync quite easily. With a `vector`, you will always know that the `size` is correct.

The second key difference from an array is that . If you create a vector with four elements in it but later discover that you need five, the vector can resize itself automatically. The internal mechanism for this isn’t simple (see assignment 6), but looking at a `vector` from the outside, it’s automatic.
Usage

Header file:

```
#include <vector>
```

Declaration:

```
std::vector<int> numbers;
std::vector<BankAccount> accounts;
std::vector<Student> students;
```

Using a vector is quite straightforward. First, you must `#include` the standard library header file it’s defined in (`<vector>`).

To create a `vector` object, simply declare it like any other variable: first the __________ and then the __________. The examples from this slide show three vectors: one vector of integers called `numbers`, one vector of `BankAccount` objects called `accounts` and one vector of `Student` objects called `students`. 
Type parameter:

`vector<int>`

All elements of same type!

The type name inside the angle brackets is called a *type parameter*. The example declares that this particular vector contains integers, not bank accounts, students or any other type.
Accessors:

```cpp
vector<int> numbers = /* ... */;

// C++98-style iteration:
for (size_t i = 0; i < numbers.size(); i++)
{
    cout << numbers[i] << "\n";
}
```

C++11 range-based iteration:

```cpp
for (int value : numbers)
{
    cout << value << "\n";
}
```

The next few slides show all of the methods of `std::vector` that you will be expected to remember. The `size` method can be used to determine how many elements the vector is currently holding. It returns a value of type `size_t`, which is an unsigned integer type (created with a `typedef` in the standard library header `cstddef`).

We can also iterate over a vector using the new C++11 range-based for loop. This style of iteration saves a little bit of syntax, but more importantly, it puts the focus of the code on the __________ ________________ rather than the mechanics of ________________. The implementation of range-based for loops is discussed further in lecture 19.
The **push_back** method adds an additional element to a vector, increasing its size by one. This is one of the primary advantages of vectors over arrays: resizing can be automatic. When desired, the vector can also be manually resized using the **resize** method: the example above resizes the vector to hold 10 elements.

**std::vector** also implements the index operator. We will discuss how this is done in **lecture 13**, but from a consumer perspective, we can use square brackets to access individual elements of the vector just like we would with an array. If a vector contains 10 elements, accessing element 10, 11, 12, etc., is a programmer error and could lead to unexpected results (e.g., a segmentation fault that terminates the program or, worse, modifying memory that doesn’t belong to you).
In C++11 and later, vector initialization can look a lot like array initialization: a comma-separated list of values within curly braces.

The standards committee that developed C++11 put significant effort into improving the consistency of initializing containers, structures and even classes.
C++98:

```c++
using std::vector;

void foo()
{
    vector<int> numbers;
    numbers.push_back(1);
    numbers.push_back(2);
    numbers.push_back(3);
    numbers.push_back(4);
    numbers.push_back(5);
}
```

Earlier versions of C++ (e.g., C++98) did not provide a convenient mechanism for initializing vectors. Instead, it was common to either `push_back` (as shown in this slide) or else to `resize`. 
Summary

Containers

- places to store multiple things

std::vector

- standard library container
- iteration

mutation
Topic 10:

Operators
We are already familiar with some operators like +, -, *, / and %. When we look more closely at them, we can see that they don’t always behave in the same way. For example, addition of integers is handled differently from addition of floating-point numbers. However, there is an even bigger difference with non-numeric types: the + operator is used for both ____________ (of numbers) and ____________ (appending strings to each other). ______________. These two operations are logically related, so it makes sense for them to share an operator, but appending one string to another is fundamentally different from adding two numbers.
Operators

What about my objects?

```cpp
enum class Currency { CAD, EUR, GBP, USD };  
class Money
{
    public:
        Money(double amount, Currency);
        Money(unsigned int whole, unsigned int sub, Currency);
        Money(const Money&) = delete;

        Currency currency() const { return currency_; }
        unsigned int valueIn(Currency) const;
    /* ... */
};
```

This concept can be taken even further when applied to user-defined types. In __________ ____________ Money.h ____, we have defined a class to represent money. Objects of the Money class can have different currencies and amounts (and, as discussed in lecture 12, it can’t be copied). It would be very natural to want write expressions such as subtotal + tax + tip, but the compiler doesn’t know how to add objects of arbitrary user-defined types.
What about my objects?

```cpp
Money pocketChange(5, Currency::CAD);
Money houseDeposit(15000, 0, Currency::CAD);
Money leftToSave = houseDeposit - pocketChange;
```

In this code example, we attempt to subtract two amounts of money, but again, the compiler doesn’t know about any operation that can apply to the − operator with two `Money` operands.
Can define your own operators:

```cpp
class Money {
   public:
      /* ... */
      Money operator + (const Money&) const;
      Money operator - (const Money&) const;
      /* ... */
};
```

Fortunately, C++ allows us to define new operations for existing operators. This ability is common to many languages (just a few examples: C++, Python, Ruby, Rust and Scala) but is not available in others (e.g., Java). It’s actually a pretty controversial issue, because it’s a feature that’s easy to overuse. For instance, would you know how to evaluate the expression \((1 \sim \%/# 2)\) from the Scala graph library? I wouldn’t!

In C++, we define operators by defining functions and methods named, e.g., `operator+` or `operator<<`. \_______________\, where \(n\) is the number of operands that the operator takes. **Binary operators** are operators that apply to two operands, e.g., \(a + b\) or \(a \times b\). **Unary operators** are operators that apply to one operand only, e.g., \(-x\) or \(-p\). \_______________\: the current object itself is one of the operands.

\_______________\ what do the two consts mean?

\_______________\ `const Money&` means that the method promises not to modify the reference passed to it as an argument. The final `const` means that the method promises not to modify the `Money` object it’s called with respect to.
Subtracting money

```cpp
Money pocketChange(5, Currency::CAD);
Money houseDeposit(15000, 0, Currency::CAD);
Money leftToSave = houseDeposit - pocketChange;
```

__________ money-example.cpp

In this code example, we use operators declared in `Money.h` to perform simple arithmetic with `Money` objects instead of plain old numbers.
Implementation:

```cpp
Money Money::operator + (const Money& other) const
{
    /* ... */
}
Money Money::operator - (const Money& other) const
{
    /* ... */
}
```

The implementation of an operator method looks ____________________, except
that its name is a bit special: the operator keyword followed by the operator we're defining a new
operation for.
This table shows two additional operators that can have different meanings depending on what types they are applied to. We have used `<<` together with `cout` and, less frequently, `>>` with `cin`, but these operators were originally used in C for bit-shift operations on integers. For instance, `(3 << 2)` means the number 3, shifted two bits to the left, which evaluates to 12 (00000011 becomes 00001100).

This is just another example of ________________________________.
Which operation is this?

```
int aBigNumber = (1 << 5);
std::cout << "Hello, world!\n";
```

It depends on the operands' types!

Providing multiple type-dependent definitions is called *overloading*

---

Given the table on the previous slide, how can we answer the question, "which operation is this?"

The answer is the same as the answer to every other really good question: "it depends!" In this case, it depends on the __________________________. The same operator (<<) can mean completely different things, depending on the types of the operands it’s operating on.

This technique (changing the meaning of something based on its type) is called *overloading*. In C++, ________________, even special ones. For instance, a class can have multiple constructors, as long as they all have parameters of different types. When we define a type-specific operator implementation, therefore, it is called *operator overloading*.
Operators

<< and >> are operators like any other:

```cpp
int aBigNumber = (1 << 5);
ostream& o = (std::cout << "Hello, world!\n");
ostream& o2 = (o << 42);
ostream& o3 = ((o2 << "Pi is") << 3.1415926);
```

The << and >> operators, then, aren’t magical: they are binary operators just like all of the other ones. Just like other binary operators, expressions involving << and >> are an expression that can be evaluated to an ostream reference. This reference can then have the << operator applied to it as we print another value, and so on. cout << x << y << z is just an expression with three operations, each of which will evaluate to an ostream&.
We can define << and >> operators:

```cpp
class Money {
    public:
        /* ... */
};
std::ostream& operator << (std::ostream&, const Money&);
```

We can also overload the << operator by defining a new operator<< function with two parameters: an ostream& and whatever type it is that we want to be able to print.

__________ why isn't this operator overload method?

__________ the object on the left isn't Money, it's an ostream. In order to overload the operator as an ostream method, we would need to change the std::ostream class itself, which isn't ours to change (it's in the standard library).

__________ Money.cpp

Now try running money-example.cpp.
Exercise:

class Student
{
    public:
    Student(const std::string& name, unsigned int id);
    std::string name() const { return name_; }
    unsigned int studentNumber() const { return id_; }

    private:
    const std::string name_; 
    const unsigned int id_; 
};

std::ostream& operator << (std::ostream& , const Student&);

As an exercise in operator overloading, ________________ for this Student class. In order to test it, you'll also need to implement the constructor and write a main function that uses it.
Operator overloading works by function overloading. In fact, we can call these overloaded operator functions explicitly as operator<<: see explicit-call.cpp for an example. Choosing which operator is run, then, is the same process as choosing which function to run: we look to see which types match.
Can also overload methods:

```cpp
class Money {
    public:
        /* ... */
        Money operator + (const Money&) const;
        Money operator + (unsigned int) const;
        /* ... */
};
```

Since class methods can also be overloaded, we can overload operators with multiple meanings using multiple methods, too. In this example, we provide an operation that can add two `Money` objects together and another that can add a scalar integer to a `Money` object.
Exercise:

```cpp
class Money
{
    public:
    /* ... */

    // Implemented in example code:
    Money operator + (const Money&) const;
    Money operator - (const Money&) const;

    // Implement:
    Money operator + (unsigned int) const;
    Money operator - (unsigned int) const;
    /* ... */
};
```

In this exercise, you should implement the `operator+(unsigned int)` and `operator-(unsigned int)` methods, possibly using `operator+(const Money&)` and `operator-(const Money&)` as examples. You may also want to modify `money-example.cpp` to test your work.
Summary

Operators
Overloads
Topic 12

Correctness and error handling
Nobody's perfect:

TicTacToe game;

game.place(Player::X, 0, 0);
game.place(Player::O, 1, 0);
game.place(Player::X, 0, 1);
game.place(Player::O, 1, 1);
game.place(Player::X, 0, 2);
game.place(Player::O, 1, 2);

**Question:** what's the problem with this code?

**Answer:** the game keeps going after X has won!

Logic may allow for a kind of perfection, but software rarely does. Software authors will always need to contend with __________________________. In this case, it's clear that something has gone wrong, but it's not entirely clear what to do about it.
Nobody's perfect:

```cpp
std::string filename = gui::OpenFileDialog();
std::fstream document(filename);
// do some work with the document
```

This slide shows another kind of error: user error. Suppose that a user asks a program to open a file, but that file doesn't actually exist. What then? Clearly something is wrong, but we haven't yet seen how to handle this kind of problem.
Nobody's perfect:

- `Money total = Money(5, Currency::CAD) + Money(5, Currency::GBP);`
- `Money total = savings.balance() + taxReturn.refund();`
- `Money total = savings.balance() + wireTransfer.amount();`

This is another example of another kind of problem that can occur in real software. Here, we are attempting to add values of different units together: Canadian Dollars and British Pounds. Now, it might be possible to convert one into the other and add those together, but we cannot directly say that five dollars plus five pounds makes five anything.

The above, slightly contrived, example might seem slightly more realistic when we can’t immediately see what currencies we are using. If we are simply adding an account balance to a tax refund or to a wire transfer amount, then it’s not immediately obvious whether we are adding values of different currencies together.
# Dealing with the unexpected

## Sources of surprise:
- Programmers
- Users
- Computers
- The outside world

## Layers of defence:
- Programmers
- The compiler
- Invariants and assertions
- Error handling

There are lots of reasons that our programs might have to deal with surprising inputs and situations. We need to deal with different categories in different ways.

First, __________________________. Much of software programming language design is about preventing mistakes. Programming languages allow programmers to express their ideas safely: we could program a **computing machines** with assembly language and get the same results, but that would require much more effort and be much more __________________________.

Second, __________________________. If software depends on user input, it must accommodate surprising inputs. Users mistype things, ask software to do things it can’t do and press "Cancel" buttons at unexpected times. These may not all be "errors", but they are challenges.

Third, __________________________. Sometimes memory, disks, etc., work as we expect them to. Other times, we run out of memory, lose a disk or have the plug pulled out of the computer. This challenge increases in the mobile context, where Bluetooth turns on and off, GPS signal comes and goes, etc.

Fourth, __________________________. Programmers sometimes get away with assuming that the local computer won't fail, but *something* fails at some point. Wireless networks can be flaky, other computers go down for maintenance and hackers cause dramatic slowdowns of major websites. Software that interacts with the outside world must be able to adapt to such failures.

The good news is that we have multiple layers of defence against error.
Dealing with the unexpected

Peer program policing:

- code review
- documentation
- specifications

One defence against programmer error is _________________. Serious software
development is almost always done as part of a team, and effective software development teams use
______________ to enhance the quality of their software. When code review is done well,
everyone checks their ego at the door and looks for opportunities to improve the code they’re
looking at. Programmers compare software against specifications, documentation and other code.
Peer review can help spot errors, but:
It's better to prevent mistakes than to catch them!

Can't compile error? Can't run it!

(Use the source, Luke!)

Code review can be enormously helpful, but even better than ____________ mistakes in implementation is ____________ them from ever occurring.

Many forms of programmer error can, when using the right language, be detected by the compiler. Many of the improvements in C++11 over C++98 are about making it easier to write correct code and making it harder to write incorrect code.
This slide shows a simple example of a programmer error. In this case, the programmer has written code that would modify the value of something whose value should not be modified (the value $\pi$). This is a surprisingly easy error to make.

One very straightforward solution to this problem is to ________________ ________. If we declare that the value of $\pi$ is a const, then ________________ ________. 
Programmer error:

```cpp
class Course
{
    public:
        /* ... */
        Course(int number, /* ... */, int capacity = 100);
        /* ... */
};
Course programming(-3891, Course::Level::Undergraduate, -50);
```

Compiler "seat belt":

```cpp
Course(unsigned int number, /* ... */, unsigned int capacity = 100);
```

The same technique (use the compiler and the type system) applies to numbers. If an integer value should never be negative, we can use the `unsigned` keyword to ensure that it never will be.
Errors

Conditions and invariants

Preconditions and postconditions:

\[ \langle x \geq 0 \rangle \quad x := x + 1 \quad \langle x > 0 \rangle \]

Invariants:

\[ \forall t \cdot x \geq 0 \]

Another defence against error is to write down a set of conditions and invariants: logical statements that ought to be true when a program runs. Simply thinking rigorously about the logic of our software tends to improve its quality, but we can also check these logical statements when we actually run the code.

We can express preconditions and postconditions for any unit of software, but we often express them in the documentation of functions and methods. A precondition is a statement \[ \text{precondition} \]. The code may not execute correctly if its preconditions are not met. A postcondition is a statement of \[ \text{postcondition} \]. Together, the pre- and post-conditions provide a logical contract for some code: "if you give me some acceptable value of \( x \), I will return you some acceptable value of \( y \)."

Invariants are statements that ought always be true. More precisely, \[ \text{invariants} \]. For instance, an object invariant is a logical statement that should be true after any method runs, assuming it was true before the method ran. In this way, invariants can be seen as both pre- and post-conditions.
Preconditions and postconditions

```cpp
class Student {
    /* ... */
    /**
     * Register the student in a course.
     *
     * @pre student is not yet enrolled in 'c'
     * @post course list includes this student
     */
    void RegisterInCourse(Course& c);
    /* ... */
};
```

Conditions and invariants are often written down in the documentation of the code they describe. This is a place where programmers who use the function/method/etc. can be exposed to things like pre-conditions. In fact, there are automated tools like Doxygen and Sphinx that will extract such comments into web pages (see, e.g., the libgrading docs).
Best-case scenario:

- prove that your program is correct
  - it **always** does the right thing
  - it **cannot** do the wrong thing

In an ideal world, we would be able to __________ these logical properties of our software. If we could prove that our code always acted correctly and never caused an undesired outcome, there would never be software "bugs": computers would work the way we expect and neither engine control units nor radiation therapy machines would kill their users.
There are some cases in which it is possible to logically prove properties of programs. This is an area of advancing research, with the world's leading experts attempting to advance the state of the art. We have now reached the point where researchers can prove useful properties about an operating system microkernel or even a small kernel.

Nonetheless, this kind of work is very difficult, requiring many person-years of work to formally verify even small amounts of code.

Part of the reason for this difficulty is that _______________ ____________, in ways that cause specific challenges to formal verification techniques (this will make more sense after we talk about pointers). Programming languages often expose functionality to users based on what is convenient or expeditious, rather than what will make the proofs easier. So, while formal verification is an important and growing area, it will be some time before this effort is usefully expended on software developed by "ordinary" software authors.
The next best thing:

- extensive testing
- assertions
  - can't prove it at compile time?
  - check it at run time!

If we can't prove software correct, it is useful to do the next best thing: ____________. By testing software with a variety of inputs, it can be possible to build confidence that the software is "mostly correct" even without formal proof (example: https://allendale.engr.mun.ca/jenkins/job/Loom/BUILD_TYPE=Release/). This isn't true for all kinds of software (e.g., concurrent software) or all kinds of applications (e.g., computer security), but when combined with coverage analysis tools, it's often as good as we get.

Another technique for detecting programmer errors is to combine testing and general usage with assertions. We may not be able to prove our conditions and invariants in the general case ("for all possible executions of this program, ..."), but can may be able to ________________ ______________________________.
The C standard library provides us with a header called **cassert** and a mechanism for checking conditions and invariants. We can check some condition (e.g., \( x < 10 \)) with the code `assert(x < 10)`. When the program runs, the standard library will check whether the condition \( x < 10 \) is true. If it is, the program keeps running normally.
This slide shows an example of an assertion failure. The previous slide had a post-condition: after the `Student::RegisterInCourse` method finishes running, the current student should be contained in the course’s class list.

On this particular run of the program, this post-condition is not met. Something has gone wrong, so the program stops running and prints the above error message. This is called a "fail-stop" approach to error handling, and is an appropriate response to programmer errors.
Aside: proprocessor definitions

```c
#include <myheader.h>
/* ... */
#if defined(WINDOWS)
    filename = "C:\Program Files\MyLibrary\my-header.h";
#elif defined(LINUX)
    filename = "/usr/include/my-header.h";
#elif defined(BSD) || defined(MAC_OS_X)
    filename = "/usr/local/include/my-header.h";
#else
    #error Unknown platform!
#endif
```

`assert()`, although it looks like a function, is not actually a function: it is a __________ __________. The C preprocessor is part of a C (or C++) compiler that can be used to make the same source code compile differently in different situations. For example, in the example on this slide, the line of code that assigns a value to the `filename` variable will be different when compiling for Windows, Linux, BSD or Mac OS X, etc.

This proprocessor munging of source code can be used to configure software in lots of different ways, not just for different platforms. One of the points of customization is _______ `assert()` _______.

Assertions:

```c
#if defined(NDEBUG)
    #define assert(condition)
#else
    #define assert(condition) \n    if (not condition) { /* report the error */ }
#endif
```

The C preprocessor is used to customize the behaviour of `assert()`. When the preprocessor definition `NDEBUG` exists, `assert()` is defined to do nothing. Otherwise, `assert()` follows the check-and-abort described on previous slides of this lecture.
Errors

Assertions:

$ clang++ -D NDEBUG main.cpp Student.cpp -o registrar
$ ./registrar
Proceeding merrily on our way with invalid data!
Here’s your course list:
< some invalid data >

rule of thumb: leave assertions on

Using the preprocessor, it is possible to disable assertions: just pass the `-D NDEBUG` argument to the compiler at the command line. Disabling assertions will mean that all checks are "compiled away to nothing". This speeds up code execution, but it also means that we miss spotting programmer errors that `assert()` would have caught.

Because assertion-checking is so fast (typically just a matter of comparing a couple of numbers) and the value of error-detection is so high, it’s often best to leave assertions enabled. Before disabling them, _____________________ measure the time it takes to run the program with and without assertions. In many cases, you will find that you cannot measure a difference.
Errors

Assertions:

- used to catch **programmer** errors
- not for user or other run-time errors!

Assertions are handy things, but they are only used for catching _______________. It is inappropriate to use an assertion to catch a run-time error. The reasons for this should become clear on the following slides.
Suppose you write a program that, in the course of its duties, opens C++ source files. You might ask the user, "what file would you like to open?" If the user chooses a file that isn’t a C++ source file, is the right response to shut the program down and print out an error message?

(______________ following the Journalistic Principle, no)
A user might be quite surprised if, after trying to open a file, the program simply closed. This is why assertions are not the most appropriate mechanism for handling user error (or any other unexpected user behaviour).
Not for run-time changes:

There are other kinds of run-time errors that can occur. This is increasingly true in a mobile world, where network connections, GPS signals and even pieces of hardware (mice, USB sticks, etc.) can come and go while your program runs. Also, as applications become increasingly interconnected with network components, .

In almost all of these cases, an assertion is not the most appropriate response to error. Even a GPS app shouldn’t abort when the GPS signal is not present: it might be better to use old data or display a message to the user such as that shown above. To do this, we will need another form of error handling.
Run-time or user errors:

- detect errors
- report errors
- handle errors

When we encounter a run-time error, we need to handle it more explicitly. Our code needs to do some extra work to make sure that we can detect, report and/or handle errors.
Explicit error handling

C-style error handling:

```c
int fd = -1;
while (fd < 0)
{
    filename = promptUserForFilename();
    fd = open(filename, O_RDONLY);
}
```

One form of explicit error handling is ________________________. In this form of error handling, ________________________. For example, when the range for a value is the set of positive integers, the number -1 can be used to represent "an error has occurred". Alternatively, a function can use pass-by-reference to effectively "return" two values: a success value (true/false) and, if successful, the actual value.

The logical consequence of this method of representing errors is that __________________________

______________________________.

This approach is used extensively in real C-based systems such as the UNIX operating system and its derivatives (BSDs, Linux, Solaris, etc.). The example on this slide is inspired by real file-opening code, but the same pattern is found in other functions used to interact with the operating system.
Explicit error handling

Simple example:

```java
double squareRoot(double x);
```

What's the worst that could happen?
Simple example:

```c
double squareRoot(double x);
```

What's the worst that could happen?

**Exercise:**

write a `squareRoot` function that reports success/failure as well as the actual square root (when successful)
Exceptions are an error-handling mechanism in C++ (and lots of other languages, too) that allows us to handle errors in a less intrusive way. When a problem is detected, instead of handling the problem right away, our code can simply flag the problem by throwing an exception. We throw an exception with the `throw` keyword, followed by a value.

----------------------------- an integer, a string or any other kind of object.

**Bonus note:**

*After covering inheritance, it will be a good practice to throw exception objects that are instances of `std::exception`.*

If we don’t handle the exception (see the next slide), `throw` behaves a lot like `return`: we leave the current function immediately. Any statements after the `throw` will not be executed.
Who should handle this error?

Whoever can fix it:

```cpp
try {
    database = connectTo(/* ... */);
    registry.use(database);
    registry.lookupStudent(id);
} catch (/* ... */) { /* reconnect! */ }
```

The other side of exceptions is what to do with them when they have been thrown. In order to handle an exception, we surround some code that could throw exceptions with a `try` block: the `try` keyword followed by a block of statements enclosed in braces. This block is then followed by `catch` blocks, which describe how to handle exceptions of different types.

One of the more elegant aspects of exceptions is that they allow programmers to decouple ___________ from ___________. When the first code example on this slide detects that the database connection is not active, what can it do about it? This method doesn't have access to the server name, username, password, etc., that might be required to re-connect to the database. All this method can do is give up: it's not capable of looking up a student unless it has an already-open database connection.

The appropriate place to handle this error is not in the depths of `Registry::lookupStudent()`. Rather, the place to re-open the connection is the place where we originally opened the connection in the first place. This is the code context where we have access to whatever database information is required to open the connection. Using `try/catch` allows us to ________________________________ rather than wherever we happen to notice it.
Exceptions

Multiple ways to handle exceptions:

```cpp
try {
    studentId = gui.prompt("What is your student ID?");
    courseId = gui.prompt("What is the course number?");
    student = lookupStudent(studentId);
    course = lookupCourse(courseId);
    student.register(course);
} catch (std::exception& e) {
    gui.reportError(e.what());
} catch (std::string& message) {
    gui.reportError(message);
}
```

When an exception occurs within the `try` block, execution immediately jumps to the `catch` block that catches the correct type and carries on from there. In this example, if `lookupStudent()` threw a `std::exception` (say, because the user entered an invalid student ID), the flow of the program would skip over `lookupCourse()` and `Student::register()`. Instead, it would jump directly to the first `catch` block, the one that handles a `std::exception`. 
Exceptions propagate through the call stack.

**Recall:**

*The call stack is a representation of how one function called another, which called another, etc., to get from `main` to any particular place in our code.*

In the code example shown here, the `Registry::lookup()` method detects that some exceptional condition has occurred. It reports this problem by throwing an exception. Since there is no `try/catch` block in this method, the exception causes execution to exit the method, like a `return` statement would: we end up back in `Connection::processInput()`. There is no `try/catch` in that method either, so the exception keeps propagating: back up to `WebServer::run()` and then finally to `main()`.

_______________ what happens when there is no `try/catch` in `main()` either?

_______________ we shall see, two slides on!
Catch exceptions by type:

```cpp
try {
    studentId = gui.prompt("What is your student ID?");
    courseId = gui.prompt("What is the course number?");
    student = lookupStudent(studentId);
    course = lookupCourse(courseId);
    student.register(course);
} catch (InputException& e) { /* ... */ }
catch (DatabaseException& e) { /* ... */ }
catch (...) { /* ... */ }
```

We can have multiple `catch` blocks attached to one `try`. Each `catch` should catch a different type of exception: when an exception is thrown, we will immediately jump to the first `catch` that can apply to the type.

______________ when could multiple types apply?

______________ in the case of implicit conversions or inheritance (which we’ll see later in the term).

If none of the `catch` blocks have an appropriate type, we can use the `catch-all` (`catch (...)`) to catch any exception. This is commonly employed as a last resort in the `main()` function or within an application development framework (e.g., your OS’ GUI library).
If nobody catches it:

```
libc++abi.dylib: terminating with uncaught exception of type char const*
```

If an exception propagates all the way to `main()` and isn’t caught there, the program will terminate. This is a bit like the behaviour of `assert`: the programmer has failed to write code to catch all of the exceptions that the program can throw. There is no good way to recover from this condition, so we treat it as a programmer error and terminate the program.
Summary

Errors

Exceptions
Pointers
Topic 13

Pointers
L-value

Shares storage with another L-value
Referential transparency:

You can treat a reference to something just like the thing itself.

Referential transparency is an important property of references. It makes references convenient, easy to use and easy to understand. ______________, there are reasons we sometimes need to work with another, older style of L-value sharing.
Older way to share an L-value

Crucial programming concept

Before C++ brought references to the fore, C programmers used *pointers* to share L-values. References are a more direct way of sharing L-values and have supplanted pointers in many use cases. However, there are still some ways in which pointers are essential, and there is even more code that uses them because they are familiar to the code’s authors. Thus, it is essential that a Foundations of Programming course cover pointers.
Recall that the usual picture of memory that we show actually imposes structure that isn’t there. The way that we draw boxes or put space between values helps us to look at and think about memory use, but real memory is just a sea of 1’s and 0’s. The computer can’t tell, by looking at a particular place in memory, that we’re storing an int vs a double vs a std::string there.
This is a slightly more realistic view: ones and zeroes with no discernible structure. Memory locations don't actually have names like b, s and x, but they do have addresses: numbers that identify a place relative to the beginning of memory (like an array index).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 0001</td>
</tr>
<tr>
<td>1</td>
<td>0000 0011</td>
</tr>
<tr>
<td>2</td>
<td>1110 1000</td>
</tr>
<tr>
<td>3</td>
<td>0100 0000</td>
</tr>
<tr>
<td>4</td>
<td>0000 1001</td>
</tr>
<tr>
<td>5</td>
<td>0010 0001</td>
</tr>
<tr>
<td>6</td>
<td>1111 1011</td>
</tr>
<tr>
<td>7</td>
<td>0100 1101</td>
</tr>
<tr>
<td>8</td>
<td>0001 0010</td>
</tr>
<tr>
<td>9</td>
<td>1101 1000</td>
</tr>
<tr>
<td>A</td>
<td>0100 1010</td>
</tr>
</tbody>
</table>
Address of an L-value

```c
short int i = 0;
int *j = /* address of i */;
```

We say:

\( j \) points to \( i \)

A pointer is simply _____________ . In the example shown here, \( i \) is a value stored at address 1. \( j \) is a _____________: the value \( j \) contains a number, and that number is the address of \( i \).
The first pointer example on this slide shows the \textit{address-of} operator. This operator ($\&$) is used to find the address of a value: \underline{\textbf{_______________}}. In this example, the address of \texttt{johnDoe} is used to initialize the variable \texttt{tallest}, which is of type “pointer to Student” (Student*).

The second example shows the \textit{dereference} operator. When $\ast$ appears in an expression like this, it is the dereference operator, which is used to visit the address indicated by a pointer. An expression such as $\ast \texttt{tallest}$ evaluates to a reference to the memory that \texttt{tallest} is pointing at. That is, it \underline{\textbf{_______________}}.

A (hopefully) helpful summary table:

<table>
<thead>
<tr>
<th>Used in type</th>
<th>Used as operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ast$ The type is a pointer.</td>
<td>Dereferences a pointer.</td>
</tr>
<tr>
<td>int*: a pointer to an int.</td>
<td>$\ast$p: turn pointer $p$ into a reference.</td>
</tr>
<tr>
<td>$&amp;$ The type is a reference.</td>
<td>Takes the address of a value.</td>
</tr>
<tr>
<td>int&amp;: a reference to an int.</td>
<td>$&amp;$i: get pointer to i.</td>
</tr>
</tbody>
</table>
Another key operator related to pointers is the *structure dereference* operator.

You may recall that the *structure reference* operator is used to access members of structures and classes. For instance, if we have a `Student` object called `johndoe`, we can call the `Student::ask` method on `johndoe` with the code shown above on the slide.

The *structure dereference* operator does the same thing, but starting from a pointer rather than a reference. That is, if we have a `Student*` (a  a `Student`) called `tallest`, we can call its `ask` method with the structure dereference operator as shown in the third code snippet.

```
student-example.cpp
```
Arrays:

```c
int someNumbers[] = { 42, 17, 54 };
int *numberPointer = someNumbers;
int *firstNumber = numberPointer;
int *nextNumber = numberPointer + 1;
```

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>42</td>
</tr>
<tr>
<td>0x1001</td>
<td>17</td>
</tr>
<tr>
<td>0x1002</td>
<td>54</td>
</tr>
</tbody>
</table>

Understanding pointers allows us to truly understand another topic that has previously been somewhat mysterious: arrays. When considering arrays in previous courses, we’ve seen that _______________, but not had the basis to understand why. We can now see the reason: _______________.

Since arrays are really pointers, we can create a pointer to an array and initialize it with ______ __________, as in the second code snippet on this slide. This pointer will point to the beginning of the array, which is _______________.

One slightly surprising facet of pointers is the way that arithmetic operations work with them. For instance, in the last code snippet on this slide, incrementing a pointer to address 0x1000 (the address of the first element in the array) does not evaluate to 0x1001, but to 0x1004! This surprising behaviour is called pointer arithmetic.

______________ arrays.cpp (see value of nextNumber!)
Unlike conventional integers, which increment and decrement by intervals of 1, pointers increment by intervals of ___________. For example, an int* value, when incremented by one, will actually increase by sizeof(int) bytes, which is 4 B on most computers today. The reason for this is that incrementing a pointer should cause the pointer to _____ _____ __________, not the next byte.

Pointer arithmetic means that there is an equivalence between pointers and arrays. Using an index in square brackets to "look up" an array element is exactly equivalent to __________________________.

This is part of why the official C++ reference says that arrays are evil.
Pointers to pointers to ...

```c
int i = 0;
int *j = &i;
int **k = &j;  // or &&i
int ***l = &k;  // or &&&i
assert(i == *j);
assert(i == **k);
assert(i == ***l);
```

Accessing values through pointers is called *indirection*: it is less direct than accessing the value, well, directly. This is more complicated than using references, but sometimes the complexity is essential to represent real problems.

It is possible to be more indirect than a single pointer: we can have pointers to pointers, pointers to pointers to pointers or however much indirection we like. Double-indirection is not uncommon in pure C code, although in C++ we would often accomplish the same task with a reference to a pointer. Higher levels of indirection are rarely, if ever, used in practice.
A pointer is a promise

Dereference i:

There is an integer at address 0x1010, please go fetch it for me and tell me its value.

Structure dereference s->ask(q):

There is a Student at 0x7fff51376800, please call its ask(std::string question) method.

When we dereference a pointer (either using the dereference or the structure dereference operator), we ___________. For instance, if we dereference an int*, we are telling the computer that "there is an integer at the address this pointer points at, now go get it." Put another way, when we dereference a int*, _____________ int ____________.
Question:
What if there isn't an integer at address 0x1010?

Answer:
The computer tries to access it anyway!

If we are mistaken, and the value at the address we're pointing to isn't actually a value of the type we thought it was, the computer can't tell. Remember, the memory is just a sea of 0's and 1's, with no inherent structure: the only structure is what we impose on it. If we make a mistake, telling the computer the wrong structure, it can't do anything but follow our directions.

invalid-pointers.cpp

The consequences of dereferencing an invalid pointer (a pointer that doesn't point at what we think it points at) vary depending on the nature of the error.
Invalid pointers

Common pointer errors:

- null pointers
- uninitialized pointers
- out-of-bounds pointers
- dangling pointers

We will cover four common kinds of invalid pointer. There are key differences among them, but ultimately they have the same core feature: ____________________________

__________________________ .
Null pointers:

```cpp
int *i = nullptr;
cout << *i << "\n";
```

More realistic example:

```cpp
Student *s = course.topStudent();
s->presentAward(/* ... */);
```

```cpp
Student* Course::topStudent() const
{
    if (students.empty())
        return nullptr;
    /* ... */
}
```

The first common pointer error is a *null pointer*. This is simply a __________. Dereferencing a null pointer will cause the computer to access the memory at address zero, which is almost certainly not the value you were looking for. In fact, this is such a common error that most operating systems tell the computer’s hardware not to let you access address zero: if you do, your program will terminate with a segmentation fault (Unix and Unix-like systems), a "program X has stopped working" message (Windows) or, for the classicists, a General Protection Fault.

The first code example on this slide may seem implausible, but it becomes more realistic when you consider the second and third examples. In these examples, there is a method of the Course class that returns a pointer to the top student in the course. If the course has no students, however, there is no valid pointer we could return. Instead, it would be common to return nullptr. If we aren’t careful, it would be very easy to dereference this nullptr.
Null pointers:

```cpp
Student *s = course.topStudent();
if (s)
    s->presentAward(/* ... */);
```

```cpp
Student& Course::topStudent() const
{
    if (students.empty())
        throw std::exception("no students in course!");
    /* ... */
```

To avoid dereferencing the null pointer, we can check it using a boolean conversion. In an if condition, a null pointer will evaluate to false and any other pointer will evaluate to true, so we can be sure that the pointer is non-null before we dereference it. Of course, that's a very ___________ ___________ ______.

In new C++ code, we might prefer to make `Course::topStudent()` return a reference to the top student and ______________________ if there is none. That way, if we receive any return value from `Course::topStudent()`, we can trust that it is, in fact, a valid `Student`. 
Invalid pointers

Common pointer errors:

- null pointers
- uninitialized pointers
Invalid pointers

Uninitialized pointers:

```cpp
int *i;
cout << *i << "\n";
```

A potentially more problematic, but in practice easy to fix, pointer problem is the *uninitialized pointer*. Just like any other integer value, if we declare a pointer variable but fail to initialize it, it could contain any value. This might happen to be zero (a.k.a., `nullptr`), so dereferencing it would terminate our program, but it could just as easily be a pointer to a valid address in our program — just not the location we were expecting.

Using an uninitialized pointer can result in invalid results (if reading) or silent data corruption (if writing). Fortunately, however, it is easy to fix by ____________ and modern compilers can warn programmers about the use of uninitialized variables.
Invalid pointers

Common pointer errors:

- null pointers
- uninitialized pointers
- out-of-bounds pointers
Another classic pointer error is the *out-of-bounds pointer*. In this example, we attempt to access an element that is one past the end of an array. This location in memory is likely to contain some other local variable, so modifying what we think is just another array element could result in unrelated variables being overwritten. This type of problem can be surprisingly difficult to debug, as seemingly unrelated variables can all start changing to unexpected or even invalid values.

In computer security terms, a maliciously-induced out-of-bounds pointer access is a [buffer overflow](https://en.wikipedia.org/wiki/Buffer_overflow). These are very real attacks that are still discovered in real software, despite enormous effort to eliminate them.
Dangling pointers:

```c
int* stackPointer()
{
    int x = 42;
    int y = /* ... */
    return &x;
}
```

One of the most difficult pointer problems to detect is the dangling pointer. This is a pointer that
__________________________. The code example on the slide above shows one
simple example: returning a pointer to a stack-allocated variable. The problem is that, when we
return from the stackPointer function, the result is a pointer to a place in memory that used to
contain the variable x but is no longer reserved for this purpose. If we try to dereference this pointer,
__________________________. If we
write to the memory at this address, we could overwrite memory that is now being used for another
purpose (e.g., a local variable in an entirely unrelated function). However, because the memory
might still have the same value, the kind of pointer problem can be particularly difficult to spot.

__________________________ dangling-pointer.cpp
A related problem:

```cpp
class Course {
    /* ... */
    unsigned int number_;  
    Level level_;          
    unsigned int capacity_;  
    std::vector<Student> students_;  
};
```

```cpp
class Course {
    /* ... */
    Level courseLevel;
    int capa;
    int numOfStudent;
    int remaining;
    int courseNum;
    std::vector<Student> students;
    std::vector<std::string> stu;
};
```

One very practical example of the kinds of problems we can have with pointers comes from the failed auto-grading of assignment 3... last night!

In this auto-grading, your code #include’d your header files as you might expect. However, the test code #include my solution’s header files. This led to a serious problem...
A real pointer problem

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>name</td>
</tr>
<tr>
<td>1011 0010</td>
<td>1011 0010</td>
</tr>
<tr>
<td>0110 0001</td>
<td>0110 0001</td>
</tr>
<tr>
<td>1100 1001</td>
<td>1100 1001</td>
</tr>
<tr>
<td>0111 1011</td>
<td>0111 1011</td>
</tr>
<tr>
<td>1001 0000</td>
<td>1001 0000</td>
</tr>
<tr>
<td>0100 1000</td>
<td>0100 1000</td>
</tr>
<tr>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0111</td>
<td>1101 1110</td>
</tr>
<tr>
<td>1101 1110</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0000 0000</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

If we were to look at a Student object in memory, we could point to the places where it stores various fields like its name and ID.

However, we should also recall that ___________________________: the memory has no labels saying, "here is the ID", we just interpret the memory that way. Crucially for this problem, that interpretation comes from ____________________.

If some code interprets memory in one way ("the student's ID is here") and other code interprets the same memory in a different way ("the student's ID is over here"), we might end up doing something nasty like ____________________________ string! This is, in fact, a behaviour that I observed when I ran some of your failing tests in a ____________.
Relevant lessons from pointers:

A pointer is a promise

Do you know what you're really pointing at?

Don't write code at 2am.

Even though this assignment didn't explicitly use pointers, there are several lessons from our "pointer problems" section that are applicable. Firstly, we must remember that a pointer is a promise. When we apply an interpretation to some memory, we are promising that the memory contains the type we claim it does. If it doesn't, ____________________________ (unless we ask it to explicitly in some very specific ways we'll see soon).

Therefore, we need to be very sure that _____________________________. In my case, that meant fixing my test scripts to ensure that I always compile with the same type declarations (i.e., the same header files). Within a single program, that means we need to exercise a lot of care when using pointers. Over the next few lectures, we'll see some techniques that will help us use pointers more safely.
Common pointer errors:

- null pointers
- uninitialized pointers
- out-of-bounds pointers
- dangling pointers
Pointers summary

Pointers are addresses

Pointer arithmetic

Pointer/array equivalence

Pointer errors

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>
The memory in any computer is, of course, much larger than the simplistic pictures that we’ve been looking at so far. Even the picture on the right, with 256 B of memory, is nothing in the great scheme of things: the computer you’re reading this text on likely has several gigabytes of memory (the computer I’m typing on has 33.5 million times the depicted amount). In practical use, we organize this vast quantity of memory into several ____________________________.

**Just for information:**

*The mechanisms by which the operating system and hardware create and manage these regions will be explored in Microprocessors (4862), Computer Architecture (6861) and Operating Systems (8894).*
Memory regions

The store

The stack

The heap

The three memory regions that software uses to store different kinds are ___________________________ (for global values), ___________________________ (for local variables) and ___________________________ (for dynamically-allocated memory).
a.k.a. "static store", "the store"

```c
int globalVariable = 42;
int main(int argc, char *argv[])
{
    int *ptr = &globalVariable;
    return 0;
}
```

The data segment, a.k.a., the "static store" or simply "the store", is ___________________________ ________. There variables can be accessed by any function, including functions running at the same time on different processors. This can cause logical problems and is one of the reasons that ____________________________. Still, when there really is a need to have exactly one instance of a variable, no matter how many objects the program creates, functions it calls or threads it spawns, global variables are still used.
a.k.a. "static store", "the store"

As you can observe in lab 4, the addresses for variables in the data segment are the same as their offsets in their object files (and quite close to the addresses of the functions defined in the program). This is because ______________________________________________________________________________________________________________________________________________________________. When the operating system loads your program, it puts the whole executable file into memory, including the spaces for global variables.
a.k.a. "static store", "the store"

```c
int globalVariable = 42;

int main(int argc, char *argv[])
{
    int *ptr = &globalVariable;
    return 0;
}
```

---

```
regions.cpp
```

Run this code example as-is to print out the addresses of a couple of global variables. Try adding more global variables and comparing their addresses. Can you demonstrate how pointer arithmetic works with the addresses of these variables? If you uncomment lines 26 through 31, __________
```
greeting
```
___________?
The stack is a memory region used to store local variables. Local variables, you may recall, are variables inside functions (or methods!). Every time you call a function, it needs space to store its local variables, independent of any space used by other functions (even, in the case of recursion, from space used by other calls to the same function!).

In the first (slightly silly) example on this slide, the `increment` function needs enough space to store two local variables when it is called: the parameter `x` and the variable `y`.

In the second code example, the `calculator` function also needs space to store two local variables: `int currentValue` and `string s`. Note that both are local variables, even though one is at the top of the function scope and one is nested within a `while` loop.
The stack is intimately related to the call stack. The stack can be represented by a pointer to a location in memory, which can be adjusted up and down to make room for variables (allocate stack memory) or to reclaim memory that we’re finished with (deallocate stack memory). When we call a function, the current position on the stack is adjusted to make room for whatever local variables that function has. For example, when `calculator` calls `doMath`, the stack pointer will be moved by enough space to make room for two `int` variables.

```cpp
// regions.cpp
```

This time, uncomment lines 33 through 45 and observe the addresses of these local variables. How far apart are these variables on the stack? Is there any relationship between these distances and the types of the variables? You can observe similar effects in `lab 4`.
void processBigMatrix()
{
    int values[100000000];
    /* ... */
}

void processBigMatrix()
{
    int *values = new int[100000000];
    /* ... */
}

The stack is a convenient memory region, as it can allocate and deallocate memory automatically as required. However, the stack is bounded in size: one would not want to create a very large array on the stack.

Exercise: try allocating increasingly-large arrays on the stack. How large an array (in bytes) can you create before you encounter problems? What error do you see when the array gets too large?

We might not want to allocate a 400,000,000 B array on the stack, but modern computers can absolutely work with 400 MB data sets. Instead of allocating them on the stack, however, we allocate them on the heap. The heap is a memory region in which we can ______________ memory when we have ______________:

- very large allocations
- allocations with special lifetime requirements (automatic allocation/deallocation isn’t appropriate)

regions.cpp

Uncomment lines 48 through 60 and explore the pointer values that new gives.
Heap allocation

Two new operators:

- new
- delete

Heap memory is managed with two operators: new and delete. We explicitly __________ new __________, effectively asking the C++ library, "please allocate me enough memory for an int, or an array of doubles, or a BankAccount object." When we are finished with the memory that was allocated for us, we explicitly ______________ delete ____________, effectively telling the C++ library, "I'm done with this memory, release it for someone else to use."
Heap allocation is done with the `new` keyword and a type. The code snippet on this slide shows allocations of a single value, an array and an object. In the last case, we are passing the value 10 to the House constructor as the object is created.
Allocation:

```cpp
double bigMatrixOperation()
{
    // Represent 10,000 x 10,000 matrix
    double *matrix = new double[100000000];
    double value = 0;
    /* ... */
    return value;
}
```

Memory leak!

When we explicitly allocate memory, _______________________. Unlike stack memory, which is automatically reclaimed when we exit a function, nothing will reclaim heap-allocated memory unless we do it ourselves. That is, __________ new ____________________________

```cpp
delete __________.
```

If we fail to deallocate memory that we explicitly allocated, the C++ library will never know that it's available for the use of others. In the code example on this slide, we allocate 800 MB and then we never deallocate it. Every time we call this function, we carve out 800 MB that can never be used by any other function. This is called a memory leak. We won't be able to call this function many times before our program runs out of available memory.
Deallocating heap memory is done with the `delete` keyword. To use this keyword, we write a statement `delete value`, where `value` is a pointer to some memory that was allocated on the heap. The only caveat is that, when deallocating an array, we use square brackets between `delete` and the array/pointer value.
Deallocation:

```c
House *myHouse = new House(10);
/* ... */
delete myHouse;
/* ... */
delete myHouse;
```

If `new` without `delete` is a problem (memory leak), so is `delete` without `new`, or even too many `deletes`. When we attempt to `delete` memory that was never allocated with `new`, or, equivalently, we attempt to `delete` memory more than once, the standard library will typically cause our program to abort. This is an example of a serious programmer error, so causing the program to fail-stop is often the appropriate response.

The use of the terms `malloc` and `free` in this message come from the C functions used to allocate and deallocate heap memory. The C++ `new` and `delete` operators are actually built on top of these underlying C functions (much as the C++ language is built on a C foundation).
This diagram depicts four memory regions, using actual addresses as set up by Mac OS X on a 64b computer (__________ sizes are not to scale!). In addition to the three regions just discussed for holding data, there is also a fourth region that is used to store code. When a compiler translates source code (see lab 2), it translates it into instructions that the computer can directly execute. These instructions are loaded into memory in the code segment, which, like the data segment, is loaded from the executable file the compiler generates.
Memory regions

- data segment ("store")
- stack
- heap
- (+ code!)

Explicit allocation
- new and delete
Q: What are methods?

Q: How do they differ from functions?

Recall that methods are __________________________. This means that ________ __________________________ using their names directly, just like calling a function or accessing a variable.
Methods are:

- defined in a class
- apply to an object

Which object?

**this** object!

Methods are defined in a *class* but refer to an *object*. For instance, the `Student::name()` method (defined in the `Student` class) can be used to retrieve a name from a `Student` object. The class has __________________, but when we call the method, it will return __________ ______________: the objects that represent Alice, Bob, Carole, etc., will all return different names.

So, the question is: when the same method runs on different objects, how can the method know which object it is running with respect to? For instance, if `Student::name()` returns the value of the current object’s `name_` field, how can it know what the current object is? Asked differently, _______________ are we running with respect to?

The answer is, **this** object!
**this**

- C++ keyword (or Java, or C#, or Javascript ...)
  - other languages: `self` or `self`-ish (Go, Python, Rust...)
- pointer to a method's object

```cpp
Student alice = /* ... */;
alice.name();
```

**this** is a C++ keyword that allows methods to refer to the __________. This keyword also makes an appearance in C++-inspired languages such as Java, C#, Javascript, etc.

Even languages that lack a **this** keyword have a similar concept. **Go**, **Python** and **Rust** all have a special way of referring to "the current object", even though they use different keywords like **self** or even allow you to use whatever name you like.

In C++, the **this** keyword is a __________. It is only available inside methods, as only methods have "a current object". In the code example shown on this slide, when running `Student::name()` on the the `alice` object, **this** will be a pointer to `alice`. 
We implicitly use `this` every time we access another member of the current object. For instance, in the `Student::name()` method depicted on this slide, access to the `name_` field of "the current object" is equivalent to accessing the `name_` field of `this` object.
Use this to avoid confusion:

```cpp
bool Matrix::operator == (const Matrix& other) {
    if (this->length != other.length)
        return false;
    /* ... */
}
```

```cpp
void Foo::addData(const double data[], size_t length) {
    this->length += length;
    /* ... */
}
```

Sometimes we use this to avoid confusion, to be clear and explicit about this vs "that other object". In the first code example on this slide, we use this to contrast with other. We could simply check whether length != other.length, but the use of this makes this distinction clearer to programmers reading our code.

In the second code example on this slide, we use this to distinguish between a field named length and a local variable (in this case, a parameter) with the same name. This time, the confusion we are preventing is not just in other programmers, it is also in the compiler: we need to distinguish two things with the same name. Of course, modern compilers can be convinced to warn you when you use the same name for fields and local variables (try using the `-Wshadow` command line argument to the compiler).
Use this to refer to the current object:

```cpp
Student& Student::changeName(const std::string& newName)
{
    name = newName;
    return *this;
}
```

Method chaining:

```cpp
Student alice(/* ... */);
/* ... */
alice.changeName(/* ... */)
    .changeGrade(/* ... */)
    .completedCourse(/* ... */)
```

One thing that we can do with this is to generate a _________________.
Since this is a pointer to the current object, we can use it to get a reference to the current object.
All we need to do is dereference this pointer in an expression that applies the deferences operator * to this, i.e., *this.

Returning a reference to the current object is a common pattern with methods, ________________
____________ (this is a ____________ for assignment 5!). When a method returns a reference to the current object, it becomes possible to immediately call another method in a practice called
______________. Method chaining often allows us to write less verbose, more readable code, as shown in the example on this slide, which comes especially handy with operators (e.g.,
cout << x << y << z).
Visibility:

defining your secrets

Friendship:

opening up your secrets

We have previously looked at visibility keywords in class declarations (so far, public and private). These keywords are used to define class members that should be kept secret, i.e., only accessible to other class members.

We will now consider the opposite case: using special C++ keywords to grant extra access to private data.
One style:

```cpp
class Student
{
    public:
    Student(/* ... */);
    std::string name() const;
    private:
    const std::string name_;
};
```

As an aside, it’s worth noting that the placement of `public` and `private` members is about style and convention rather than correctness. As in most style matters, whether you put `public` members before `private` or vice versa doesn’t matter. What matters is that you __________ ________________.

The example shown here uses the style that I use in all of the 3891 code examples: `public` members first (starting with constructors) and `private` members last. This convention ________________, i.e., developers who `use` the class rather than `implement` it.
Another style:

```cpp
class Student
{
    private:
    const std::string name_; 

    public:
    Student(/* ... */);
    std::string name() const;
};
```

Another common style is to put all `private` members first in a class declaration. This style
Another style:

```cpp
class Student
{
    public:
    std::string name() const;

    private:
    const std::string name_;

    public:
    Student(/* ... */);
};
```

It is also possible to have lots of visibility keywords: one for each member if you like. These can be in any order: the language doesn’t specify one correct way. However, I would suggest that you have some governing principle behind why you choose the visibility keyword orderings that you do!
Aside: a point of style

```cpp
const std::string name_;
```

VS

```cpp
const std::string m_name;
```

VS

```cpp
const std::string myName;
```

On another side point, it is common practice in C++ to apply some prefix or suffix to fields in a class. In this course, I tend to use an underscore suffix, but other common conventions include an `m_` or `my` prefix. Once again, as this is a style issue, the important thing is not so much which convention you follow as that ______________________________. This helps make code more readable, as other developers will know what to expect.

The reason for this convention is that, in C++, ______________________________. That is, if you have a field named `name`, you cannot also have a method called `name`. Other languages, such as Java, have special rules that allow fields and methods to re-use each others’ names, but C++ does not: if a field and a method had the same name, the compiler could get confused (and, frankly, so could many developers!).
Back to the point:

private members can only be accessed by methods of the same class

Returning to the main point, you should recall that private members (both fields and methods) can only be accessed by methods of the same class. All other code (functions or methods of other classes) is forbidden from accessing private members.
We can write:

```cpp
struct StudentID {
    const unsigned int year;
    const unsigned int serial;
};
std::ostream& operator << (std::ostream&, const StudentID&);
```

You should also recall that, in general, operator overloads can be written either as functions or as methods. However, in some specific cases (e.g., on this slide), we can only write an operator as a function. This operator inserts the contents of a `StudentID` into a `std::ostream`, and is perfectly legal to write.

_______________ why is this a function rather than a method?

_______________ LHS is an `ostream`, not `StudentID`

*Code example:* [Student.cpp](#)
We can't write:

```cpp
class Student
{
    /* ... */
    std::string name() const { return name_; }

private:
    const std::string name_;  
    const StudentID id_;  
};

std::ostream& operator << (std::ostream&, const Student&);
```

Like the code example on the previous slide, this operator must be written as a function rather than a method. Unlike the previous case, however, this function cannot be written as shown. The reason is that the `Student::id_` field is private, and `Student` provides no accessor method (e.g., `id()`), so the insertion operator has no way to access the student’s ID. That is, _______________ ______________________________. This is the role of C++ friendship.
Friendship

**IRL:**

someone to whom you tell your secrets

One practical definition of a friend in real life is someone you trust: someone you are willing to share, among other things, secret information with.
In C++:

something with access to `private` fields and methods

In C++, the term `friend` has a narrower definition. A `friend` in C++ is something that can access the private members of a class.
Two kinds of friends:

functions

```cpp
class Foo {
    /* ... */
    friend void printFoo(Foo&);
};
```

classes

```cpp
class Foo {
    /* ... */
    friend class Bar;
};
```

C++ friends can be functions or classes. A class can extra grant access to a friend function with the keyword `friend` followed by the declaration of the function. This usage of `friend` is shown in the first code example on this slide. After this declaration of friendship, the named function will be accessible to class `Foo`.

Friendship can also be declared with other classes. If class `A` wants to give class `B` access to its private members, we need to declare the ____________ with `friend class B` inside the declaration of class `A`. If class `B` wants to reciprocate and give `A` access to the private members of `B`, `B` needs to make its own declaration: `friend class A`.

*Code example: Student.h / Student.cpp*

Take this code example and use `friend` to help you write a complete and correct `operator << (std::ostream&, const Student&)`. 
Methods are:

- defined in a class
- apply to an object

Recall that (ordinary) methods are defined in a class but apply to an object (which object? this object!). However, there is another type of method that, while defined in a class, __________ ________________.
The `static` keyword can be used to specify that a method is __________. That is, the compiler can figure out how to run the method without needing to know anything about any objects of (in this case) class `Student`. In fact, the method ________________ ________________.

Static methods, because they are static, are called with the `scope resolution operator (::)` rather than the `structure reference operator (.)`. When called, they ________________.
Given that static methods act a lot like functions, it’s a legitimate question to ask: "why not just use functions?"

In fact, static methods are a lot like friend functions. They are statically-named units of code that have access to the private members of a class (e.g., a private constructor) or an object (e.g., fields and methods). The difference is really about naming.

While a friend function has access to private members, a static method is more obviously tightly bound to a class, since it’s declared and defined inside of it. For example, if we were to create a friend function that can create a Student object, we might call it CreateStudent. We might then also have CreateCourse and CreateTerm functions. This isn’t a terrible naming system, but the names Student::Create, Course::Create and Term::Create make it clear that these methods are really parts of classes, and there’s no purpose for them apart from these classes.

________________________ why would we use a Create method rather than a constructor?
Bad:

```cpp
SeismicMatrix::SeismicMatrix(size_t n) : matrixData(new double[n * n]) {}
```

Better:

```cpp
SeismicMatrix SeismicMatrix::Create(size_t n) {
    double bigMatrix[] = new double[n * n];
    if (not bigMatrix)
        return nullptr;   // or throw an exception
    return SeismicMatrix(bigMatrix, n);
}
```

__________ what's wrong with this?

__________ the constructor can fail!

You may recall that constructors don't have any great way of reporting errors. Constructors don't return values, so we can't propagate an error code, and ____________ (some style guides prohibit it, some software teams disable exceptions entirely and other situations just require care to prevent memory leaks).

A better way to implement the same logic is to do the "heavy lifting" in a static `Create` method, which can fail in any convenient way. In this slide's second code example, a failure in memory allocation (e.g., running out of memory) can be detected and/or the exception can be propagated without worry.
this
friend
static
Topic 11

Iterators
Iteration is the process of working with a collection of things, one thing at a time. This is a very common thing to do with arrays...
One thing at a time:

```c
int sum(int *values, size_t length)
{
    int total = 0;
    for (size_t i = 0; i < length; i++)
        total += values[i];
    return total;
}
```

... which, as we saw in lecture 16, is the same thing as iterating over values pointed at by a pointer.
One thing at a time:

```cpp
int sum(vector<int>& values)
{
    int total = 0;
    for (size_t i = 0; i < values.size(); i++)
        total += values[i];
    return total;
}
```

We can use the same pattern when iterating over values in a vector...
One thing at a time:

```c++
int sum(vector<int>& values)
{
    int total = 0;
    for (int value : values)
        total += value;
    return total;
}
```

... or, as we saw in lecture 10, we can use C++11’s range-based for loop.
Not just vectors:

```cpp
std::vector<int> values = { /* ... */ };  
for (int x : values) { /* ... */ }
```

```cpp
Player tictactoeBoard[] = { /* ... */ };  
for (const Player p : tictactoeBoard) { /* ... */ }
```

```cpp
std::list<Student> students = /* ... */;  
for (Student& s : students) { /* ... */ }
```

```cpp
std::map<std::string, int> names = /* ... */;  
for (auto& i : names) { /* ... */ }
```

This pattern applies to much more than just arrays and vectors, however. In fact, we want to be able to iterate over all kinds of containers, whether they are arrays, standard library containers or objects we’ve designed to hold some values.
In some cases, all we need to iterate over a set of values is a pointer. If we have an array of contiguous values (values that are next to each other in memory), we can iterate over them via pointer arithmetic.
Not just vectors:

```cpp
std::list<Student> students = /* ... */;
for (Student& s : students) { /* ... */ }
```

In other cases, iteration is more complicated. In a linked list (not a data structure you're responsible for in this course, just a useful example), every element has a pointer to the next element. The mechanism is more complicated, but the end result is the same: we iterate over all of the elements in the collection.
Not just vectors:

```cpp
std::vector<int> values = { /* ... */};
for (int x : values) { /* ... */ }
```

```cpp
Player tictactoeBoard[] = { /* ... */};
for (const Player p : tictactoeBoard) { /* ... */ }
```

```cpp
std::list<Student> students = /* ... */;
for (Student& s : students) { /* ... */ }
```

```cpp
std::map<std::string, int> names = /* ... */;
for (auto& i : names) { /* ... */ }
```

Iteration, then, is something that applies to all containers. Even more, we can use C++11 range-based for loops with all containers from the standard library (as well as arrays). No matter what type of container we're using, the iteration looks basically the same.
What all of these different types of iterators have in common is that they allow us to work with **one thing at a time** and they have some internal notion of **where we currently are in the iteration**. That is, we can ask, "what's the current value?" and "have we reached the end?"
An iterator, then, is anything that can iterate through the elements of a container and produce references to those elements. To satisfy this requirement, we need to be able to do three things, as shown on this slide.

**Go to the next thing:** we need to be able to increment the iterator, moving from the current element to the next one.

**Give me the current thing:** we need to be able to convert an iterator into a reference to the current element.

**Are we done yet:** we need some way to check whether an iterator is at the end of a container.
Given the requirements for an iterator expressed on the previous slide, we can now see that one very simple form of iterator is a pointer. A pointer can be incremented (i++), dereferenced (*i) and compared with an "end" value (e.g., i == array + size). There are much more complicated iterators, where we need to instantiate objects of iterator classes with methods that follow pointers to the next element, etc., but sometimes all we need is a pointer.
Iterators

Forward

Bidirectional

Random-access
A forward iterator is one that is capable of iterating in one direction only. In the example on this slide, the iterator for a linked list can move to the next element in the list, but there is no way to move backwards.

**Once again:** you are **not** responsible for understanding, describing or using linked lists or any other container type that isn’t std::vector, but these examples are helpful for understanding different iterator categories.
A bidirectional iterator, as the name suggests, can be both increment and decremented. We can move both forwards and backwards through the container with this category of iterator. The example on the slide depicts a doubly-linked list, in which each element has a pointer to both the next element and the previous element. This allows the iterator to move forwards or backwards by one element.

Since a bidirectional iterator can do everything a forward iterator can do (and then some), we can say that a bidirectional iterator is a special kind of forward iterator.
Iterators

Random access:

- "Go to the next thing"
- "Go to the previous thing"
- "Go forward/backward N spaces"

The final category of iterator that we will discuss is the *random access iterator*. A random access iterator can move forwards or backwards, like the bidirectional iterator, but it can do it by an arbitrary number of elements in a single operation. This means that a random access iterator is a special kind of bidirectional iterator. For example, in order to increment the bidirectional iterator from the previous slide by ten, we would need to increment the iterator ten times. With a random access iterator, we can perform a single "move ahead ten spaces" operation.

We have already seen and worked with random access iterators: a *pointer is a random access iterator*. We can increment or decrement it an arbitrary number of spaces using *pointer arithmetic*. 
Iterators in `std::vector`:

```cpp
vector<int> numbers = /* ... */;
for (int x : numbers) { /* ... */ }
```

// Iterator pointing at the first element
vector<int>::iterator i = numbers.begin();

// Iterator pointing just past the last element
vector<int>::iterator j = numbers.end();

`std::vector`, like all standard library containers, provides two methods that return iterators: `begin` and `end`. `begin`, as you might expect, returns an iterator to the beginning of the container. `end` returns an iterator that is just past the end of the container. This is perhaps less intuitive, but it allows us to check whether we've reached the end of a container with the simple test `i == container.end()`.

**Bonus note:**

Standard library containers can have more methods that return iterators, such as `rbegin/rend` (which return iterators to walk through the container backwards) and `const` versions that return iterators with a `const` promise, i.e., you cannot use them to modify the container.
For loop with iterators:

Replace:

```cpp
for (size_t i = 0; i < values.size(); i++)
    useValue(values[i]);
```

With:

```cpp
for (vector<int>::iterator i = values.begin(); i != values.end(); i++)
    useValue(*i);
```

We can replace C-style iteration (start at index 0, access each element in turn until the index is equal to the size of the array) with iterator-based iteration as shown on this slide. One advantage of this change is that the new form of iteration will work even on containers that don't support indexing (i.e., `values[i]`). One disadvantage is that the type of our iterator variable tends to get long and, sometimes, complex.
Iterators

For loop with iterators:

Replace:

```cpp
for (size_t i = 0; i < values.size(); i++)
    useValue(values[i]);
```

With:

```cpp
for (auto i = values.begin(); i != values.end(); i++)
    useValue(*i);
```

```cpp
for (int value : values)
    useValue(value);
```

We can simplify this form of for loop by using a new keyword that is available to us in C++11 and up: auto. The auto keyword means, let the compiler figure out what the type is. In the second example on this slide, the variable i will still have a type, which will still be vector<int>::iterator as on the previous slide, but we don't have to type it ourselves. This can be convenient, as sometimes the details of the iterator type actually get in the way of reading the code: what other programmers reading this code really need to know is that i is an iterator over values, and a long type name can get in the way of seeing this. This is especially true when iterator types get very complicated, e.g., std::map<std::string, std::list<int>>::const_iterator.
If you define:

```cpp
class SomeClass
{
    public:
        typedef /* ... */ iterator;
        iterator begin();
        iterator end();
};
```

You can use:

```cpp
for (/* ... */ value : someObject)
{
}
```

When a container provides `begin` and `end` methods, C++11 will allow us to use an even clearer form of the for loop: the **range-based for loop**. In this form, we don’t have to specify anything about the iterator that we use to iterate over the container: the compiler generates all of those low-level details for us. This works whether the container is from the standard library or we wrote it. **Your types can (and should) provide** `begin/end` **to allow range-based iteration.**

**Code example:** [IntArray.cpp](IntArray.cpp)
Summary

Iterators

Range-based for loops

New keyword: auto
We have been talking up to this point about some common problems that come up with pointers. This slide includes two examples: a ________________ and a ________________. In the first example, we return a pointer to a stack address when the variable being pointed out is going out of scope. This means that some code, somewhere, will have a pointer to a location that used to store a variable we care about. However, that memory might contain something very different now!

In the second example, we have allocated some memory on the heap with new but we never deallocate that memory with delete. This means that the memory is still reserved for our use, even though we are done with it. This kind of leakage causes our programs to reserve more and more memory over time, even when they don’t need it, which can eventually cause the whole system to run out of memory!

We would like to be able to prevent both of these kinds of problems.
Obvious lifetime:

```cpp
double bigMatrixOperation()
{
    // Represent 10,000 x 10,000 matrix
    double *matrix = new double[100000000];
    double value = 0;
    /* ... */
    return value;
}
```

Something that both of these examples have in common is that both memory allocations (one on the stack, the other on the heap) both have fairly obvious lifetimes: we can point at the places in the code where we _________________. In the first case (the stack allocation), we are finished with the variable _________________.

The second case, despite using a different memory region (the ___________ instead of the ___________), actually has a very similar lifetime. A bit like a stack variable, we allocate the memory somewhere in the function and finish working with it at the end of the function.

The second code example on this slide shows the fix for our memory leak. It is very simple: we need to de-allocate the memory at the end of the function. On the stack, this would be handled automatically for us, but because we're using the heap we need to _____________.

Still, at the end of the day, we essentially want to treat this particular memory allocation like ____ ______________.
While the previous slide had a very obvious lifetime for the heap-allocated memory, not all situations are that straightforward. The example on this slide shows a registration system that needs to manage students, courses and registrations. In this case, the registration system will create lots of Student objects on the heap, since they need to live for a long time. These objects must be ______________ at some point... but when?

**Question:** Why not check Student arguments in `Student::Student()`?

**Answer:**

There may be no good error-handling options: constructors can’t return error values and many style guides (e.g., Google and LLVM) do not allow the use of exceptions.
This slide shows function calls (solid black arrows) and pointer usage (dashed orange arrow). Pointers get initialized by the `new` operator, returned from functions and passed to other functions. All of these function may have different lifetimes, leading to a great deal of confusion about when the heap-allocated memory should be deallocated. This confusion leads to both _________ (if memory is deallocated too soon) and ________________ (if memory is deallocated too late / not at all).
Approach for managing lifetimes

- easy for pointers to propagate
- need discipline:

**One owner for each allocation**

**One lifetime for each owner**

---

As we saw on the previous slide, it is easy for pointers to propagate. Pointers get returned, copied and passed around in ways that make it easy to write code but make it hard to reason about [blank]. One solution to this problem of promiscuous pointer propagation is to impose a little discipline on our programming practice.

This discipline consists of forcing ourselves to obey two simple rules in order to simplify our pointer management problem:

1. for every heap allocation, there must be something (an object, function, etc.) that owns it and

2. everything that owns memory allocations should have a clearly-defined lifetime (which could be in terms of its owner, e.g., "the Student is deallocated when the RegistrationSystem is").
If we impose this discipline on ourselves, we can clean up the "rat's nest" of pointers that we saw a couple of slides ago. Now the ownership of memory is much clearer, although this comes at a cost: we can't necessarily just pass pointers around, we may need to talk to an allocation's ____________ in order to access the allocation.
Big idea:

- dynamic allocations "owned" by an object
- object responsible for cleaning up after itself

**Resource Acquisition Is Initialization (RAII)**

If we follow this memory ownership discipline, every allocation will have an owner and every owner is responsible for cleaning up after itself by deallocating its explicit (heap) allocations.

When these two practices are combined, we end up with a *design pattern* called RAII: resource acquisition is initialization. That is, an object that owns memory *acquires* its ownership at the time it is *initialized*. That means that if you have an initialized memory owner in front of you (e.g., you have an object as a local variable), then you don’t need to worry about leaks: the memory is owned by something that knows how to de-allocate it at the appropriate time.

(this pattern is also applicable to other kinds of resources, but we'll limit our discussion to heap allocations for now)
Recall the ideal constructor:

```cpp
SomeClass::SomeClass(/* ... */)
  : field1(parameter1), field2(parameter2), // ...
{
}
```

```cpp
class Matrix
{
  /* ... */
  size_t rows_, columns_;  // Note: No initializer here.
  double *values_;  // Note: No initializer here.
}
```

How can we ensure the `Matrix` owns its `values_`?

If we are using the "ideal" pattern for our constructors, as we've previously seen, how can we ensure that an object always owns the heap allocations it has pointers to?
Create the array in a static method:

```cpp
Matrix Matrix::Create(size_t rows, size_t columns) {
    double *values = new double[rows * columns];
    if (not values)
        throw std::string("Egad! We failed to allocate the memory!");
    return Matrix(rows, columns, values);
}
```

Private constructor takes ownership of array:

```cpp
Matrix::Matrix(size_t rows, size_t columns, double *values)
    : rows_(rows), columns_(columns), values_(values)
```

One common pattern that we saw in the last lecture is the use of static methods to create objects rather than constructors. Static methods have two ways to report errors (exceptions and error values), making them a good place to do large allocations or acquire other resources. If those allocations ___________, we can then pass all of the relevant values to the object’s _____________.

This allows constructors to be simple, just copying values into fields if possible. However, it also helps us maintain a useful invariant: that any object we could work with has been properly initialized. That is, we don’t have to wonder, "is this a valid object"? Implicitly, ________ ________ constructed in this way is a valid object.
Recall the Big Idea:

- dynamic allocations "owned" by an object
- object responsible for cleaning up after itself

Constructors can take ownership of pointers

What about cleanup?

Now we’ve seen how objects can take ownership of resources. The next question is, what about the other part of memory ownership? How can objects ____________________________?
Destructors

Constructors initialize objects

Destructors clean up objects

As the name suggests, destructors are the opposite of constructors. Where a constructor initializes the fields of an object, taking ownership of resources, a destructor cleans up the fields of an object, discharging the terminal responsibility of ownership. This is a bit like decommissioning a ship, a piece of equipment, etc., when we're done with it: it is the owner's responsibility to make sure we've put everything away safely, cleaned up any future hazards, etc.
Destructors are declared much like constructors, but with two important differences:

1. their names start with a tilde (~) and
2. they have no parameters.

**Question:** How many different constructors can a class have?

**Question:** How about destructors?

The definition of a destructor looks much like that of a constructor, but again, with a tilde in the name and no parameters. Inside the definition of a destructor, we “clean up” any allocations that we own.

**Question:** Why don’t we have to clean up rows or columns?
RAII lifetime:

- constructor acquires ownership
- ???
- destructor "cleans up" (deallocates)
Copying and assigning:

```
int x = 42;
int y = x;
Matrix m = Matrix::Create(3, 3);
Matrix copy = m;
```

On this slide, we start to look at the problem of *copying pointer fields*. In this code sample, we are relying on the ________________ generated by the compiler to copy matrix m into copy.

As the diagram shows, there is a problem with this approach: we end up with two objects the contain the same pointer, both of which will act as if ________________!
When we copy a Matrix object, we actually want to end up with the situation shown on this slide’s second diagram. Rather than two objects that both think they own the same heap allocation, we'd like there to be two objects that each own an ________________ heap allocation. That way, any changes that we make to one will have no effect on the other, i.e., they are ________________________.
Matrix m3(m1);
m3 = m2;
Matrix m4 = m2;

Copy constructor and assignment operator:

class Matrix
{
    public:
    Matrix(const Matrix&);
    Matrix& operator = (const Matrix&);
}

This problem can come up whether we are explicitly invoking the copy constructor, as in the first line of this code example, or using an assignment operator (as in the second line). These two situations are actually quite different: in one case we are initializing a new object and in the second case we are assigning new values into an existing object.

**Question:** Which is occurring in the third line?

In order to handle these two situations, we need the *copy constructor* (which we’ve seen before) and the *assignment operator* (which is new to us).
Assignment operator:

class Matrix
{
    Matrix& operator = (const Matrix&);
    /* ... */
    size_t rows, columns;
    double *values;
}

Matrix& Matrix::operator = (const Matrix& other)
{
    delete values;
    this->rows = other.rows;
    this->columns = other.columns;
    this->values = /* ... do "deep copy" of other.values ... */
    return *this;
}

The assignment operator is used to assign a new value into an existing object, which may require __________________________. It is declared as an operator that takes a const reference to an object as its sole parameter and that returns a non-const reference to itself.

The definition of the assignment operator must first "clean up" any currently-owned values and then make a copy of the object being passed in. However, in order to avoid the "double ownership" problem we saw a few slides ago, we may need to do a _______________.

Deep copy:

```cpp
this->values_ = new double[rows_ * columns_];
for (size_t i = 0; i < rows_ * columns_; i++)
    this->values_[i] = other.values_[i];
```

A deep copy means that, instead of simply copying a pointer value from object to another (this would be a ____________), we need to create our own memory allocation and then copy the content of the original allocation into it. In our matrix example, we need to allocate our own array of values and then copy all of the original matrix’s values into our new array.

By using the deep copy technique, we end up with two objects that have two independent allocations with the same content rather than double ownership.
Assignment operator:

```cpp
Matrix& Matrix::operator = (const Matrix& other)
{
    rows = other.rows;
    columns = other.columns;
    /* ... */
}
```

Simple copy constructor (not always possible):

```cpp
Matrix::Matrix(const Matrix& other)
{
    *this = other;
}
```

Depending on the details of the class, we may be able to implement the copy constructor in terms of the assignment operator, as shown on this slide’s second code example.

**Question:** When is this possible?
The Big Three:

1. Copy constructor
2. Assignment operator
3. Destructor

Help us manage ownership:

- Matrix "owns" values
- Deep copy / assign
- Destructor deletes

These three special methods are often collectively referred to as the "Big Three". Together, they help us manage memory ownership by acquiring ownership of resources (such as memory allocations) and by cleaning up those resources when we’re all done with them.
The Big Three

Copy constructor
Assignment operator
Destructor

If you need one:
you probably need all three!

The simple rule of thumb with the "big three" is that, if you need one of them to manage resource ownership correctly, you _________________.

New help in C++11:

```cpp
std::shared_ptr<Matrix>
```

```cpp
std::unique_ptr<Matrix>
```

But first... the Big Five?

*Move constructor, move assignment...*

Next time!
This slide shows an example of putting our `Matrix` copy constructor and assignment operator into practice. The second line calls a function that puts some data into a `Matrix` object called `m2`, and the third line copies that data into another `Matrix` object called `m4`.

Take a moment to look through the addition operator in the `Matrix.cpp` code example. Imagine stepping through the addition that occurs on line four of this slide's code example. How many times do we copy the resulting `Matrix` data?

This (fairly simple) example involes one statement (`Matrix sum = m2 + m4`) with two operations (`m2 + m4` and `sum = /* result */`), but this simple example involes ________ ________:

- in `operator+`: copying `*this` into `sum`,
- returning from `operator+`: copying `sum` into the R-value that contains the result of calling `operator+` (the evaluation of `(m2 + m4)`) and
- copying the data from the `(m2 + m4)` R-value when constructing `sum!`
Larger matrix:

\[ N = 10,000 \]

Fundamental complexity:

\[ N \times N = 10^8 \text{ floating-point additions} \]

Copies done:

\[ 3N \times N \times \text{sizeof(double)} = 2.4 \times 10^9 \text{ B} \]

In the previous example, adding extra data copies isn’t terribly serious because we’re working with pretty small amounts of data. However, what happens if we’re working with an

\[ N \times N \]

array?

The computational complexity of an

\[ N \times N \]

matrix addition is

\[ N \times N \]

: we can’t write a program that does fewer than

\[ N \times N \]

addition operations (and correctly adds the matrices!).

In our example, however, three copies of a

\[ 10,000 \times 10,000 \]
This wastage can be seen in `operator+`: we create a `Matrix` called `sum`, copy some data into it, operate on it and then, as we’re leaving the operator, we copy its contents into a temporary matrix before throwing away `sum` (which will call a destructor that deallocates the recently-copied memory). We will then copy the values from that temporary `Matrix` object into the `sum` variable in the `main` function, just before ________________________________!

That is *twice* that we’ve deep-copied some values from one object to another, potentially at great cost, only to immediately throw away the values from the source object!
Recall that it is important to manage the ownership of the memory: every allocation needs _______ _________, and if we break that rule, Bad Things (tm) can happen.

So, to make sure that we maintain memory ownership and prevent Bad Things (tm) from happening, we can play it safe and copy data "all willy-nilly". This is a __________ thing to do, but it is not very __________.
Move semantics

Memory ownership matters:

Moving ownership is safe and fast:

Great! How does it work?

Move semantics allow us to, instead of copying data to preserve memory ownership and its safety properties, to transfer ownership from an old object into a new object.
Move semantics

```cpp
Matrix Matrix::operator + (const Matrix& b) const
{
    Matrix sum(*this);
    sum += b;
    return sum;
}
```

What we really want to do at the end of this method is "steal" the memory allocation from `sum` for the temporary `Matrix` object that actually gets returned from the function.
In order to add move semantics to class, we need to add two more special method: a \textit{move constructor} and a \textit{move assignment operator}. The declarations for these methods is shown on this slide, but there is another bit of knowledge that we need before we will understand them.

For now, we will simply say that the double-ampersand (\&\&) symbol is a \textit{special kind of reference}. 

Will explain && in a few slides!

(for now: it's a special kind of reference)
This slide shows the first part of a move constructor at work. Ignoring the details of the double-ampersand, we can observe that the initializer list looks simpler than that of a copy constructor: it performs a __________ of fields from the original object. This means that, if we didn't have any more code in the body of the constructor, we would end up with two objects with identical fields, including the pointer to owned memory.

If this were all we had, we would violate __________. However, of course, this is not all we have in a move constructor!
A move constructor has two responsibilities. One is to make a shallow copy of the original object. However, that’s a copy, not a move. The rest of the work, the part that makes this a move rather than a copy, is to ______________. That means that we will reset any fields that represent ______________ to remove the ownership. In the case of memory ownership, that means setting pointers to nullptr. Once this has been done, the new object will own the allocated memory and the old object will own ______________.

This would be a good time to re-open Matrix.cpp and explore it some more.

But next, we need to see how does this "special reference" works.
R-value references

Review:

- L-values have storage
- R-values do not

```c
int x;
x = 42;
```

You should recall that an L-value has ________________ that we can talk about; we can ________________ or assign to it. An R-value is a literal value (e.g., 42) or a temporary value (e.g., "the Matrix returned from a method").
L-value reference:

- reference to an L-value
- shares storage with another L-value

```cpp
int x;
x = 42;
int& y = x;
```

What we have been calling just a "reference" thus far is, as of C++11, more fully described as an L-value reference. That is, it is a reference to an L-value, another name for an existing L-value.
R-value references

R-value reference:

- reference to an R-value
- provides access to temporary objects

```cpp
Matrix sum = (a + b);
Matrix::Matrix(Matrix&&) { /* ... */ }
```

C++11 introduced another kind of reference, the R-value reference. Its symbol is the double ampersand (`&&`), and it allows us to put a name on an R-value. In this slide, the result of evaluating `(a + b)` will be a temporary value, an R-value. Because it’s not an L-value, it can be passed to the _______________ via an R-value reference.

So, whenever we ______________ or ______________ from a temporary value, the compiler will use the move constructor or move assignment operator if they exist.
Converting L- to R-value references:

Matrix m(3, 3);
m.set(0, 0, 1);
m.set(1, 1, 1);
m.set(2, 2, 1);

???

Move constructors and move assignment operators can only be used with R-values: temporaries and literals. But what if we *want* to steal values from an L-value?

Matrix thief = std::move(m);

// at this point, 'thief' is a 3x3 identity matrix
// and 'm' is an empty Matrix object

In that case, we can explicitly use the std::move() function, which takes a reference to an L-value as its sole argument and returns the ____________ from the function as a temporary R-value. This R-value is then something that can be passed to a move constructor or move assignment operator.

Try re-opening Matrix.cpp and using std::move() on sum.

Of course, there is also a terrible, terribly pun here (first pointed out to me by a student in 2014): this is an example of committing ______________.
The Big Three

Or the Big Five?

1. Copy constructor
2. Assignment operator
3. Destructor
4. Move constructor
5. Move assignment operator

We have previously seen the Big Three methods: the copy constructor, the (copy) assignment operator and the destructor. Now we add two more: the move constructor and the move assignment operator. The Rule of Three is now a bit more complex: it’s the Rule of Three or Five or Possibly None!

That is: it’s possible to have copy constructor/assignment operator without a move constructor/assignment operator or vice versa, but if we need either special constructor, we probably need its corresponding assignment operator as well as a ______________.
Or the Big Nuttin'? 

```cpp
class Matrix
{
    public:
        static Matrix* Create(/* ... */);
        /* ... */
    private:
        SomeMagicTypeThatCleansUpAfterItself<double> values_; 
};
```

Of course, even simpler than the Big Three or the Big Five is the Big Zero: what if we could get away without needing either the copy or the move constructor/assignment operator combination? If we could only have fields in our classes that knew how to copy or move themselves, we could ______________________ default copy and move constructors rather than implement ownership ourselves.
The good news is that ______________ at our disposal! If our fields are primitive types, the computer already knows how to copy them and we don’t need to worry about moving them. If our fields are ______________, they will have whatever constructors and destructors they need to manage their own memory. It’s only when we incorporate *pointers representing ownership* that we need our own copy/move constructors/assignment operators and destructors. If we stick to objects (e.g., fields of type `std::vector`), we don’t need to write the ______________ ourselves.
The reason why:

```cpp
/* ... */
class unique_ptr
{
    public:
    /* ... */
    unique_ptr(unique_ptr&& u);
    ~unique_ptr();
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    /* ... */
};
```

Next time...

A large part of why we’ve been talking about R-value references and move constructors, etc., is so that we can get some help with our memory management. Rather than writing our own memory management, wouldn’t it be nice if we could use some existing, magical standard library code for managing memory ownership?

The excellent news is that ____________: the standard library includes new smart pointer classes as of C++11 that have their own handling of move semantics, allowing us to manage memory safely without writing all of this annoying code ourselves!
Smart pointers

Q: what is a smart pointer?
   A pointer with an Engineering degree

An object that "wraps" a pointer

- defines pointer-like operators (*, ->, etc.)
- additional functionality (e.g., ownership)

I know I've had more than my fair share of bad puns, but this one isn't my fault! I asked this question on a final exam a couple of years ago, and this is one of the best/funniest answers I got.

A smart pointer is an object that looks like a bit like a pointer, only better. One can perform pointer-like operations on it, such as dereferencing and structure dereferencing. However, a smart pointer can provide additional functionality, such as managing the ownership of the memory the pointer points at.
Use smart pointers like "dumb" pointers:

```cpp
unique_ptr<Image> image = Image::Blank(640, 480);
cout << "image=" << image->lines() << "x" << image->columns() << "\n";

Pixel *firstRow = image->row(0);
Pixel *secondRow = (*image)[1];
cout << "rows are " << (secondRow - firstRow) << " pixels apart\n";

Image& i = *image;
Pixel *rowTheSecond = i[1];
```

This slide shows how we can use a C++11 smart pointer object much like a regular pointer. In this code example (which you can see in the code example `image-test.cpp`), we have a pointer to an Image object stored in an object of type `unique_ptr<Image>`. We can call methods of the Image object using the structure dereference operator (→) and we can dereference the pointer, obtaining a reference to it, using the dereference operator (*).

Now we'll see how a smart pointer can be used to do more than a regular pointer.
Smart pointers

Using `std::unique_ptr`

- `unique_ptr` owns a heap-allocated pointer
  - will `delete` pointer on destruction
  - must be the **only** owner
- cannot be copied
  - can only be **moved**

The `unique_ptr` class in the standard library fulfills the requirements of memory ownership by taking a pointer in its constructor, storing that pointer in a field and by using the `delete` operator in the destructor (so you must not pass pointers into its constructor unless ________________ ________________). In order to fulfill our requirements for memory ownership, a `unique_ptr` that believes it owns a heap allocation should be, well, unique! There must only be one `unique_ptr` that points to any given allocation.

The uniqueness of a `unique_ptr` is enforced by preventing copying and by providing a ________________.

Question: what’s one way we’ve seen thus far to prevent copying objects?

Answer: ________________ `delete`
Using `std::unique_ptr`

```cpp
unique_ptr<Image> image = Image::Blank(640, 480);
unique_ptr<Image> thief = image;
```

This slide shows what happens when we try to copy a `unique_ptr`: the compiler complains that we've tried to use the copy constructor, but that the copy constructor has been *implicitly deleted*. What does this mean?
"Implicitly-deleted copy constructor"

???

Default copy constructor is deleted if:

- you explicitly delete it:

  ```cpp
  Foo(const Foo&) = delete;
  ```

- move constructor or move assignment operator

  (source: §12.8.7 of C++ standard)

We have already seen how we can explicitly delete a copy constructor: using the `delete` keyword. This tells the compiler, "please don't generate a default copy constructor", as we are creating a class to describe objects that shouldn’t be copied. Now, however, there is another way to have no copy constructor!

If we define a move constructor or a move assignment operator in our class, the compiler will not generate a default copy constructor. That is, if we tell the compiler, "this is the sort of object that should have move semantics rather than copy semantics", it will respect our decision. If we want to have both ________________ semantics, we need to explicitly create both copy and move constructors/assignment operators.
Using `std::unique_ptr`

```cpp
unique_ptr<Image> image = Image::Blank(640, 480);
unique_ptr<Image> thief = image;
unique_ptr<Image> thief = std::move(image);
```

// image's pointer has been "stolen"
if (image) { /* ... will not execute ... */ }

**Question:** what is wrong with the first code example on this slide?

**Answer:** 

**Question:** how can we fix it?

**Answer:**

```
unique_ptr<Image> image = Image::Blank(640, 480);
unique_ptr<Image> thief = std::move(image);
```

When we run this code example, after using `std::move` to move the contents of `image` into `thief`, there will be nothing left “inside” `image`. That is, this `unique_ptr` will contain a `nullptr`, meaning there is nothing to point to. Thus, when we convert to a boolean value in the `if` condition, it will evaluate to `false`. 

```
unique.ptr.cpp
```
So now our tangled tale of memory ownership has shown us that double ownership is bad (why?), copying is safe but slow (why?) and move construction/assignment is fast while preserving memory ownership. However, what about the fourth case on this slide? What happens when we legitimately need to share ownership of a resource (such as a heap memory allocation)? What would that even mean?
Ownership as responsibility

Exclusive:

- car, land
- control
- lock-out/tag-out

Shared:

- heat, light, coffee
- lock-out/tag-out (again!)

There are plenty of examples of things in which ownership is exclusive. If I/my family own(s) a car or some land, that means that you do not. This also means that I am responsible for my car, my house, etc., in ways that you are not: if my house falls on somebody and hurts them, it’s my responsibility. Exclusivity also applies to more abstract concepts like control of things like vehicles: a flight crew may have a pilot and a co-pilot, but only one of them has control of the aircraft at a time (there are even specific phrases used to transfer control: "you have control / I have control", "I relieve you sir / I stand relieved").

This principle also applies to safety-critical systems via procedures such as lock-out/tag-out. When an electrician turns a switch off in order to work on the electrical line that switch controls, that electrician has a vested interest in ensuring that the switch isn’t turned back on unexpectedly! S/he will often put a lock on the switch to ensure that nobody will flip it until it’s safe to do so and the lock has been removed.

For many other types of resources, there is a shared ownership. Which of us has exclusive interest in the lights being turned on and the heat working? When an office has a shared coffee pot, everybody has an interest in that coffee pot not being left empty. This is even true for the non-coffee-drinkers: nobody wants cranky colleagues!

On a more serious note, shared ownership also applies to lock-out/tag-out. If there are multiple workers who need to ensure that a switch, valve, etc., is not activated, they can all lock it. Only when everyone is done working on the system can it be activated.
Resource "ownership":

- responsibility
- this example: exclusive
- in general: can be shared

When we talk about owning resources, then, we are talking about responsibility for them: responsibility for their use and responsibility for cleaning up when we’re all done (e.g., removing the locks from switches when the job’s done). This ownership can be exclusive in some cases, and when we are talking about exclusive ownership of memory, we use __________________ to help us uphold the memory ownership pattern. Sometimes, however, we need a shared pattern of memory ownership: how can we enforce this pattern?
What is the C++ equivalent of the "lock box" shown on this slide? How can we have multiple shared owners who all have an interest in the ownership of a memory allocation without giving any one owner exclusive control / requiring exclusive responsibility?
std::shared_ptr

Shared ownership of a pointer

Each `shared_ptr` object:

- increments a counter when created
- decrements a counter on destruction and
- if the counter is 0, deletes pointer

We can use objects of the `std::shared_ptr` class to represent shared ownership of a memory allocation. In this model, instead of memory having a single owner, there are multiple "owners" who all have an interest in the memory allocation not being de-allocated, but none has the exclusive responsibility for cleanup.
The actual implementation of `shared_ptr` involves `shared_ptr` objects that keep a reference to a common object that actually owns the memory. This object also includes a count of how many `shared_ptr` objects there are referring to this memory. The copy constructor / copy assignment operator of `shared_ptr` increments this counter, and the destructor decrements it. If decrementing the counter causes it to reach 0, nobody is using the memory any more and it can be safely ______________.
Using `std::unique_ptr`

```cpp
unique_ptr<Image> image = Image::Blank(640, 480);
unique_ptr<Image> thief = image;  // compile fails!

unique_ptr<Image> thief = std::move(image);

// image's pointer has been "stolen"
if (image) { /* ... will not execute ... */ }
```

Using `shared_ptr` can look a lot like using `unique_ptr`, with one important exception. This slide shows the `unique_ptr` usage that we saw a few slides ago, and the next one...
Using `std::shared_ptr`

```cpp
class Image { ... };

shared_ptr<Image> image = Image::Blank(640, 480);
shared_ptr<Image> anotherCopy = image;
```

```cpp
// image still has its pointer
if (image) { /* ... will execute ... */ }
```

```cpp
// Can still "steal" the ownership if we want to:
shared_ptr<Image> thief = std::move(image);
```

... shows some important differences with `shared_ptr`. This time, because the memory ownership is shared, the first block of code compiles successfully: we can make the `anotherCopy` object if we want to. This is because we aren’t copying the memory ownership, we are taking out another share in the ownership. Thus, `image` still has a share in the ownership (so the second code block executes the statements inside the `if`) and `anotherCopy` will also have a share in the ownership.

If we want to move ownership, like we have to with `unique_ptr`, that’s still allowed: we can use `std::move()` in exactly the same way. The difference is that ____________________. This means that, for the cost of maintaining some extra reference-counting complexity, we can have a more flexible memory ownership model than is possible using `unique_ptr`. 
std::unique_ptr
std::shared_ptr
This slide has an example of a pretty simple class for representing various kinds of publications (books, magazine articles, etc.). Since we have various kinds of publications that we might choose to represent, each `Publication` object must store some information that is common to all publication kinds (e.g., a title, an author) as well as some information that is only required by certain `Publication` objects (e.g., a journal name, which is only needed by objects that represent journal publications). We also need to store a value that says what kind of publication this `Publication` object represents (encoded in the `kind_` field).

*Code example:* [Publication.h](#)

*Code example:* [library.cpp](#)
Classes

Problem:

```cpp
const std::string journal_; // only for jour. article
const unsigned int volume_; // only for mag/journal
const unsigned int issue_; // only for mag/journal
```

- state we don’t need
- complex implementation methods

One problem with this approach is that we end up carrying around lots of state that we don’t need. There are fields here (journal_, volume_, etc.) that are very important for some Publication objects but not for all (or even most). If most Publication objects don’t store a journal name, it’s a bit wasteful to create space for such storage in every single object.

Even more problematic is the complexity of dealing with these various Publication kinds and implementing methods of the Publication class. If you look at the Publication::reference() method in Publication.cpp, you will see that it’s a long, complex method that needs to check, "Am I this kind of publication? Ok, then I’ll implement the method this way. If not, am I this other kind of publication? Then I’ll do it this way...". This leads to code that is complex and hard to follow, but even worse, it’s difficult to ________________.

**Question:** what if there are dozens of publication types (and there are), and we need to add another one?

**Answer:** we need to add another Publication::Kind and adjust all of Publication’s methods accordingly.

**Question:** what if you didn’t write the Publication class, but you’re relying on it via a library that you’ve licensed?

**Answer:** you might be stuck!
Classes

Some things in common

Some things not

The essential problem here is that classes are used to represent the things that objects have in common, and while there are many things that all of these Publication objects have in common, there are also many things that are not in common. So, a single Publication class isn't the best fit for this problem.
This slide shows another example of commonality and difference. Every car has the ability to tell you how fast it’s going and to turn left (hopefully it can do more than this, but this is a pretty good start!). This should be true whether we’re talking about a Toyota, BMW or anything in between: it is true ______________, it is true about the abstraction of a car.

When we get more specific, we see that certain specific makes of car will add features to our generic Car abstraction. When you buy a BMW, you are getting a luxury vehicle with a snazzy in-dash navigation system. When you buy a Subaru, you are getting All-Wheel Drive. A BMW can do all of the things that any old Car can do plus its unique, special abilities, and likewise with a Subaru.
Class commonalities

A BMW is a special kind of Car

A Subaru is a special kind of Car

We can observe, then, that a BMW is a type of Car, and that a Subaru is a type of Car, but that these types \underline{\text{are not a subclass}} to the generic abstraction that we call "Car".

In programming, we often use diagrams to represent these kinds of relationships. This slide shows a simple diagram drawn according to the \textit{Unified Modeling Language (UML)} that says a BMW class is a special kind of Car class, and that a Subaru class is a special kind of Car class too.
If we think about the problem in a different domain, we can say that any two-dimensional shape should have an area. One particular type of shape is a rectangle, which calculates its area in a certain way. We can say that a rectangle ________ shape. However, the same can be said of a square: it ________ rectangle, and it adds the particular detail that its two dimensions must have the same length.
Square is a Rectangle is a Shape:

This "is a" relationship can be applied transitively: a square is a rectangle and a rectangle is a shape, so we can say that a square is a shape. This sort of transitive "is a" relationship can be represented in the UML too, as shown in this slide's diagram.
A subclass is a special case of its superclass.

- a BWM is a special kind of car
- a square is a special kind of rectangle
- a book is a special kind of publication

This sort of "is a" relationship is expressed in object-oriented programming languages using inheritance. Inheritance creates a relationship between a subclass and a superclass, in which an object of the ___________ is an object of the ___________. This slide shows several examples of how this sort of relationship could work out in various problem domains: cars, shapes and publications.
Subclasses *inherit* superclass members:

```java
BMW dreamCar;
Subaru realCar;
double x = dreamCar.currentSpeed();
double y = realCar.currentSpeed();
```

(subclass includes the members of its superclass)

Remember: we want to describe what is *common* and what is *special* about different classes. Inheritance most directly helps us with the ____________ part of the problem: when we construct a subclass/superclass relationship, the subclass will *inherit* all of the members of the superclass. In terms of methods, that means that any methods that exist in the superclass will also exist (one way or another) in the subclass.

(we'll talk about field inheritance in just a few slides)
Subclasses *specialize* superclasses:

```java
BMW dreamCar;
Subaru realCar;

dreamCar.activateNavigationSystem();
realCar.testAllWheelDrive();
```

(subclass *adds the members that makes it special*)

On the other side of the common/special problem, the reason that we create subclasses instead of simply using the superclasses is that the subclasses ____________ their superclasses. That is, we can add things to subclasses that make it special and distinct from its superclass, allowing the subclass to focus solely on what makes it ____________ rather than what it has in ____________ with its superclass.

Put another way, when building a BMW, we don't have to re-describe all of the things that are common to all cars: we just need to talk about the "secret sauce" that makes a Bimmer a Bimmer.
Illustration in UML:

(uniﬁed modeling language)

This slide shows the graphical UML description of an inheritance relationship. Now the question is, how do express this relationship in code?
Illustration in code:

```cpp
class BMW : public Car
{
};

class Subaru : public Car
{
};
```

A BMW is a special kind of Car

A Subaru is a special kind of Car

In C++, we declare that a class is a subclass by appending: `public Superclass` to the class definition, right after the subclass name (where `Superclass` is actually the name of the superclass, e.g., `Car` or `Publication`, not the literal name `Superclass`). Other languages use similar mechanisms (e.g., `extends` `Superclass` in Java); the meaning is the same: we are creating a new class that is a subclass of an existing class.
Fields are inherited:

```cpp
class Publication {
  /* ... */
  const std::string title_
  const std::string author_
};

class Book : public Publication {
  private:
    const unsigned int edition_
};
```

Inheritance of `Publication` includes `fields` as well as methods. If a superclass has a field in it, then objects of the `Publication` will also have that field.

*Code example:* [Publication.h]
This slide shows the fields that will be contained in objects of our `Book` class: `title_`, `author_`, and `edition_`.

*Question:* where are each of these fields declared?
This slide shows where each of the fields in a Book object comes from: title_ and author_ from Publication and edition_ from Book. Note that these are all fields in every ________________: the distinction drawn here is between the reason they fields are in Book, not whether they are.
Accessing base/superclass fields:

class Publication
{
    /* ... */
    std::string title_;}

string Book::reference() const
{
    return /* ... */ + title_ + /* ... */;
}

Since our Book class has inherited a title field from Publication, we might be naturally inclined to write some code like that shown on this slide: accessing our title field directly, like any other field. This can work in some cases, but not in this particular case. Why not? We’ll address that on the next slide!
Accessing base/superclass fields:

```cpp
class Publication
{
    private:
    std::string title_;  
};
```

```cpp
string Book::reference() const
{
    return /* ... */ + title_ + /* ... */;
}
```

This slide shows the reason that our `Book` object's methods can't access its own `title_` field directly: that field was declared by `Publication` to be `private`.

**Question:** What does private mean?

Even though `title_` is a field of the `Book` class, the methods of `Book` can't access it directly! `private` really does mean `private`. If we want `Book` to be able to access the fields it inherits from `Publication`, we need a new visibility keyword that is different from both `public` and `private`.  

New keyword!

public, private and ...

```cpp
class Publication
{
    /* ... */
    protected:
    std::string title_; // Accessible to methods of this class and its subclasses
};
```

The new keyword that we need is **protected**. This keyword opens up the visibility of a class' members not just to methods of the class itself, but also to methods of _______________.

How can a Book initialize its `title_`?

```cpp
class Publication {
    public:
        Publication(std::string title_);

    protected:
        std::string title() const { return title_; }

    private:
        std::string title;
};

Book textbook(/* ... */);
```

The declaration of `Publication` shown on this slide does not make the `title_` field protected: it instead makes the field private and allows subclasses to __________ its value using a protected method called `title()`. This way, a subclass of `Publication` can observe the value of `title_` but cannot __________. This approach allows `Publication` to take responsibility for maintaining certain invariants on fields, such as "the title must not be empty".

Given this declaration of `Publication`, however, the question is: how can a subclass of `Publication` (like `Book`) initialize all of its fields? Normally we would do this in the `Book` constructor, but here is a field that the `Book` constructor is not allowed to write to!
Base constructor:

```cpp
class Publication
{
    public:
    Publication(std::string title, std::string author, int year);
    /* ... */
};
```

```cpp
Book::Book(std::string title, std::string author, int year, unsigned int edition)
    : Publication(title, author, year), edition_(edition)
{
}
```

**Question:** How do we normally initialize the fields of a Publication?

The approach we can use to initialize the Publication-inherited fields from within Book is to use the Publication constructor itself. Within the initializer list of the Book constructor, before we initialize the fields of Book, we can pass some values into the Publication constructor and let it initialize the fields we got from Publication. That is, if we got some fields from a superclass, we can let the superclass worry about how to initialize them.

This use of base constructors can be handy for two reasons:

1. we may not be able to write to the inherited fields, but also
2. we may not want to initialize the inherited fields: it could be complicated, so rather than duplicating the logic of the superclass constructor it’s better to just let the superclass constructor do the work.
So, to summarize:

1. subclasses **inherit** fields from their superclasses,

2. inherited fields are only visible to the subclass if they are **public** or **protected** and

3. we can use the superclass’ **constructor** to initialize the fields that we inherited from the superclass.
It's not just fields that are inherited from superclasses to subclasses: subclasses inherit all members of their superclasses, which includes ____________. This means that superclasses can contain methods that are common to all subclasses (e.g., returning the title or author of a Publication, since every type of publication has a title and an author). It also means that subclasses can specialize the superclass by adding new methods that are particular to them (e.g., Book::edition()) or even by replacing common/generic versions of superclass methods with versions that are more specific and appropriate to the subclass.

As an example, we might like to split up the huge Publication::reference() method from Publication.cpp by providing a common, generic version in the Publication class and then putting the "how to reference a book" version in the Book class, the "how to reference a journal article" version in JournalArticle, etc.
This slide shows what we'd like to accomplish a little more concretely. We would like to have a `Publication::reference()` method that generate a very generic form of reference (e.g., ""Care for your kitten", A Mews, 1997."), but allow subclasses to specify their own, more specific versions (e.g., ""Care for your kitten", A Mews, 2nd Ed., 978-0004125435, 1997." ) that take precedence over the generic superclass version. In order to accomplish this objective, we will need to learn two new ideas (and their corresponding C++ keywords).
New keywords: **virtual** and **override**

- **virtual**
  - meant to be specialized by subclasses
  - might have a generic, common version
- **override**
  - method specializes a superclass' method

A **virtual** method is one that is *designed* to be replaced by a subclass' version of the same method. A superclass can provide a version of the method that would not be incorrect if used by any method of any subclass, but that version may not be very specific (just like it's not *wrong* to cite a book by title, author and year alone, but it's also not the best / most specific way to do it). If a subclass provides its own version of a superclass' **virtual** method, the __________ version will take precedence: this is called ____________.

When we override a superclass' method in a subclass, it must be *exactly* the same method: the same name, the same parameters and the same **const**-ness (or lack of **const**-ness). If any of those things are different from the superclass' method, we aren't actually overriding anything: we're simply defining a new and **different** method in the subclass (and, if the name is the same, we are **overloading** the method). In order to avoid making this kind of mistake, C++11 and later provide us with the **override** keyword, which tells the compiler, "my intent here is to override a method of a superclass". If we *aren't* overriding a method, either because we missed the method name, parameters, etc., or else because the superclass' method isn't designed to be overridden (i.e., it isn't **virtual**), the compiler will helpful report an error rather than produce a program that doesn't do what we meant.
Virtual methods in action:

```cpp
class Publication {
    public:
    /* ... */
    std::string author() const;
    virtual std::string reference() const;
    /* ... */
};
```

```cpp
class Book : public Publication {
    public:
    std::string reference() const override;
};
```

We can see examples of method overriding in `Publication.cpp` (though you may need to uncomment some code).
Recall:

A subclass is a special case of its superclass.

Substitutability:

You can substitute a subclass for its superclass.

In all of this discussion of polymorphism, we are talking about how an object of a subclass is also an object of the superclass. This kind of "is a" relationship leads to an important principle in software design: that of substitutability.

The principle of substitutability says that, when we have an object of a subclass, we can substitute that object for an object of its superclass. Put another way, we find ourselves in a situation where we need an object of a superclass (a Shape, a Publication, a Car, etc.), we can satisfy that need with an object of one of that superclass' subclasses (a Square, a JournalArticle, a Toyota, etc.).
Substitutability of real-world objects:

<table>
<thead>
<tr>
<th>I ask for:</th>
<th>You bring me:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A publication</td>
<td>A book</td>
</tr>
<tr>
<td>A shape</td>
<td>A book</td>
</tr>
<tr>
<td>A shape</td>
<td>A dodecagon</td>
</tr>
<tr>
<td>A car</td>
<td>An apple</td>
</tr>
<tr>
<td>A car</td>
<td>A BMW</td>
</tr>
</tbody>
</table>
Substitutability of software objects:

<table>
<thead>
<tr>
<th>I ask for:</th>
<th>You bring me:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Publication</td>
<td>A Book</td>
</tr>
</tbody>
</table>

Publication& p = getMeABook();
cout << p.reference() << std::endl;

The substitutability of software objects is illustrated on this slide. Here, we declare a value of type Publication& (a reference to a Publication) named p. This reference must bind to some kind of Publication, and one acceptable way to satisfy this requirement is to bind it to a Book object. Then, we can deal generically with the object, forgetting that it happens to be a Book, but calling whichever of the methods are declared in Publication while trusting that any Book overrides will be called when appropriate. This kind of substitution works for any subclass that we access by ________________________.
Inheritance:

- substitutability

Fields:

- protected visibility
- base constructors

Methods:

- virtual and override
Unified Modeling Language:

This slide shows the level of detail we’ve seen in UML so far: classes have names and they can have inheritance relationships.
Class:

Three boxes:

- name
- fields
- methods

Visibility: +/−/#

This slide shows some more UML detail, and it's just about as far as we will go with this mode of representation. In UML, the box representing a class has three parts:

1. the class’ **name**,
2. the class’ **fields** and
3. the class’ **methods**.

Fields and methods can be decorated with visibility attributes:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>public</td>
</tr>
<tr>
<td>#</td>
<td>protected</td>
</tr>
<tr>
<td>−</td>
<td>private</td>
</tr>
</tbody>
</table>
This UML can be translated into the C++ shown on this slide. The same UML can be translated into similar-looking things in just about any object-oriented language, but of course we are focussing on C++ in this course.

It’s worth noting that, on this slide, I have ordered the fields and methods in a way that corresponds strongly to the UML. This is a matter of convention: some people like to put the fields first in a class declaration, like in the UML. This representation tends to emphasize the importance of a class’ ____________, i.e., the state of that class’ objects.
Class:

class Rectangle
{
    public:
    Rectangle(double l, double w) const;
    double area() const;

    protected:
    void setLength(double);

    private:
    double length_;
    double width_;
};

I, personally, prefer to put things in the order of visibility, emphasizing what’s __________ about a class before what’s __________ before what’s __________. This is a matter of style and convention: it does matter, but different groups of developers will make different decisions.
Purpose:

- to communicate design ideas
- rough sketch vs code generation

*We don't always include all of the details!*

UML (unified modeling language) diagrams are useful for communicating design ideas. You can go into a lot of detail with UML: the specification is huge, with lots of different diagram types, exhaustive enough to allow drawing diagrams from which code can be generated. However, that's a much more detailed use of UML than we will get into: we're just going to use UML *class diagrams* as rough sketches.

In fact, it’s a subject of debate as to whether or not it’s a good idea to draw diagrams so complex that you can generate code from them — I would contend that writing the code is often simpler and clearer!
Inheritance:

- open-headed arrow
- points from subclass to superclass
- "Square inherits from Rectangle"

In this course, we will use class diagrams with two kinds of arrows. The first kind of arrow is the one shown on this slide: an open-headed arrow from a ________________ to a _________________. This arrow indicates that one class ________________ from another, or alternately put, that one class is a special case of another.

It’s worth noting that the subclass/superclass relationship doesn’t depend on which class is shown as higher on the diagram: it’s all about which way the arrow is pointing.
This slide shows a class hierarchy for various types of shapes. Something that all shapes have in common is that they have ________, so it makes sense to put a virtual area() method in the Shape class. Various subclasses of Shape can override this method to allow different implementations of area(): $\pi r^2$, $l \times w$, etc.

**Question:** what is the common implementation of area() that doesn't rely on any knowledge of the shape’s details?

___________________________

There is no generic way to calculate a shape's area without knowing what kind of shape it is. So, while we might *declare* the virtual area() method in the Shape class, we cannot *implement* it there.
A method that is declared in a superclass without implementation is called a **pure virtual method** (or an **abstract method** in some other languages, like Java). We denote a pure virtual method in C++ by appending `= 0` to a virtual method’s declaration. A virtual method is __________ to be overridden, but a pure virtual method __________ be overridden.
Why?

Generality:

```cpp
vector<unique_ptr<Publication>> library;
/* ... create / retrieve some publications ... */
cout << "Library has " << library.size() << " publications:\n";
for (unique_ptr<Publication>& p : library)
{
    cout << " - " << p->reference() << std::endl;
}
```

(upcasting)

What’s the point of this? Why bother declaring a method that isn’t _____________?

The reason is that, sometimes, we don’t want to have to know the details of a particular object’s subclass. In many cases, we might want to have a container full of Publication objects or Car objects or Shape objects, no matter what specific subclass those objects are of. We wouldn’t want to have to create a separate library for every type of publication that we might want to have on our shelves!

*Code example: library.cpp*
Substitutability:

If you have a **reference** or **pointer** to a subclass, you can **treat it as its superclass** (i.e., be **more general**).

- a **BWM** is a **Car**

```cpp
BMW bmw(/* ... */);  
Car& c = bmw;
```

- a **Rectangle** is a **Shape**

```cpp
Shape *s = new Rectangle(/* ... */);
```

Recall the principle of *substitutability*: if we have a reference or pointer to a ____________, we can treat it as a reference or pointer to a ____________. We can do this without any special casting or other work.
**Upcasting**

![Upcasting Diagram]

### Example:

Using `shapes.cpp`, work through the code example(s) on this slide and determine what will be printed out by each call to the `PrintShapeInfo` function.
Abstract class:

A class with one or more pure virtual methods

(vs. a concrete class)

When a class contains at least one pure virtual method, we call it an abstract class. An abstract class is one with a piece missing: there is no implementation for the pure virtual method. Without implementations for all of the class’ methods, we cannot directly instantiate an object of an abstract class.

Classes that are complete and instantiable are referred to as concrete classes. We can create objects of concrete classes on the stack (e.g., `Foo foo()`) or the heap (e.g., `new Foo()`), as these classes are complete.
Abstract classes

Abstract class:

```
$ make
c++ -std=c++11 -Wall -g shapes.cpp -o shapes
shapes.cpp:18:8: error: variable type 'Shape' is an abstract class
    Shape s;
    ^
./shapes.h:4:17: note: unimplemented pure virtual method 'area' in 'Shape'
    virtual double area() const = 0;
    ^
1 error generated.
*** Error code 1
```

This slide shows the error message our compiler will give us if we attempt to instantiate (create an object of) an abstract class.
Abstract class:
A class with one or more pure virtual methods

Interface:
A class with only pure virtual methods

An abstract class is a class that contains at least one pure virtual method. Abstract classes cannot be instantiated because they are missing at least one method implementation. However, an abstract class may contain fields and some method implementations.

A class that does not contain any fields or method implementations is called an interface. This is a description of what a class does, not how it does it.
Recall this class hierarchy from our recent lectures. It contains a virtual method (in fact, in `Shape`, a pure virtual method) to calculate the shape’s area.
class Shape
{
public:
    virtual double area() const = 0;
    std::string name() const { return "Shape"; }
};

class Circle : public Shape
{
    /* ... */
    double area() const override { return Pi * pow(radius_, 2); }
    std::string name() const { return "Circle"; }
    /* ... */
};

Now suppose that the Shape class has two methods: a virtual method to calculate the shape’s area and a non-virtual method to return the shape’s name. Circle’s implementation of area() calculates \( \pi r^2 \) and its name() method returns "Circle".

In fact, the C-style string literal "Circle", which is a C-style string (array of char values) is automatically converted to a std::string object as the value is returned, since std::string has a constructor that takes a char* argument.
A simple class hierarchy

```cpp
class Shape {
    public:
        virtual double area() const = 0;
        std::string name() const { return "Shape"; }
};

class Rectangle : public Shape {
    /* ... */
    double area() const override { return length_ * width_; }
    std::string name() const { return "Rectangle"; }
    /* ... */
};
```

In this hierarchy, `Rectangle`'s implement of `area()` calculates \( l \times w \) and its `name()` method returns "Rectangle".
The `Square` class doesn't have its own implementation of `area()`: it relies on `Rectangle`'s implementation. All `Square` needs to do is ensure that the length and width values are the same. However, it does have its own `name()` implementation.
A simple class hierarchy

class Shape
{
    public:
        virtual double area() const = 0;
        std::string name() const { return "Shape"; }
};

Circle circle(10);
cout << circle.name() << ": " << circle.area() << "\n";

Rectangle rectangle(4, 5);
cout << rectangle.name() << ": " << rectangle.area() << "\n";

Square square(5);
cout << square.name() << ": " << square.area() << "\n";

Shape& shape = circle;
cout << shape.name() << ": " << shape.area() << "\n";

Question:
Given the code shown on the previous slides (and in `vtable-example.cpp`), what will the code on this slide output?

Answer:
The area of a Circle with radius 10 is 314.159
The area of a Rectangle with length 4 and width 5 is 20
The area of a Square with side length 5 is 25
The area of the Shape is 314.159
We have even seen (especially in Lab 4) code stored in memory. However, the question is: when we run a virtual method on a particular object, how can we find the right implementation of the method that we want to run?

This slide shows a very straightforward view of the contents of memory related to a couple of objects: there is space in memory to store our objects' fields, but also to store their ____________. For instance, the `Circle::area()` method is a set of machine instructions that, if run, will execute the instructions necessary to calculate a `Circle` object's area. That's great if we know that our object is a `Circle`, but we don't always know that! What if, for example, we have a reference to a `Shape` and we don't know which specific `Shape` subclass? Making life even more interesting, of course, while real memory does have the sorts of values shown in this diagram, it doesn't have the convenient labels!
Real memory looks a bit more like this: lots of values scattered all over the place with no inherent structure. The structure is something we overlay on top of the memory, interpreting various values according to the representations defined by their ______________. So now, given that memory looks like this, we might reasonably ask: how can we find the correct implementation of a virtual method to run? If we have a reference to a Shape and we call its area() method, how can we find the correct area() method (e.g., Circle::area())?
The answer to the question on the previous page is _____________. When a class has virtual methods, the compiler will automatically generate an array of pointers to methods (called a vtable) for that class. For example, if a class like `Circle` has two virtual methods (e.g., named `area` and `name`), the compiler will generate a vtable for it containing two entries: one for `area` (which points to `Circle::area`) and one for `name` (which points to `Circle::name`). Every `Circle` object will then have a pointer to the `Circle` vtable automatically added by the compiler.

Because the `Shape` class, in the example on this slide, has virtual methods `area` and `name`, we know that all _____________ of `Shape` must have those methods too. If we want to call the `area` method of some `Shape` object we have a reference or a pointer to, then, we can find the correct method to run by:

1. getting the object’s vtable pointer,
2. following that pointer to find the vtable for the object’s class,
3. looking up the method’s pointer and
4. following that pointer to find the correct method (e.g., `Circle::area`).
The example on this slide shows vtable use in action. Here, our Shape class has a virtual area method but a non-virtual name method (as in the example output from a few slides ago). We have a reference to a Shape object (of some kind: we don’t know what subclass), and we proceed to call its name and area methods.

When we call name, we see that we have a Shape object and that there is a (non-virtual) Shape::name method; this method gets executed even if we tried to override it in Square::name or Circle::name.

When we call area, however, something different happens. Because Shape::area is a virtual method, we do not attempt to call it directly. Instead, we use our object’s vtable pointer to find the vtable, which tells us where to find the area method by pointing to Circle::area, Square::area, etc.
**Virtual methods**

**virtual method:**

- *designed* to be overridden
- correct implementation in vtable

---

So, to summarize: a virtual method is a method that:

- is *designed* to be overridden and
- can be found and called by looking it up in a vtable.
Virtual methods

Common error:

Undefined symbols for architecture x86_64:
"vtable for Rectangle", referenced from:
Rectangle::Rectangle(double, double) in ...
NOTE: a missing vtable usually means the first
non-inline virtual member function
has no definition.
ld: symbol(s) not found for architecture x86_64

This slide shows a compiler (actually, linker) error that is commonly encountered when compiling
classes with virtual methods. We have seen **undefined symbol** errors before, when we’ve
_____________ and **called** a method but not ____________ it. However, when we forget to
implement a **virtual** method, it can cause the vtable to be missing as well as the method definition
(this usually happens when you forget to implement the **first** virtual method in a class). So, when
you see this error, start looking for missing virtual methods!
Remember `std::ostream`?

```cpp
namespace std {
    class ostream {
        /* ... */
    };

    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    extern ostream clog;
}
```

We have used `std::istream` and `std::ostream` before. These are classes defined in the `std` namespace that facilitate input and output using `insertion` and `extraction` operators. The `std` namespace also defines some values of these types: `cout`, `cerr` and `clog` are objects of type `std::ostream` and `cin` is a `std::istream`. 
Remember `std::ostream`?

<table>
<thead>
<tr>
<th>ostream</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator &lt;&lt; (bool&amp;) : ostream&amp;</td>
</tr>
<tr>
<td>operator &lt;&lt; (int&amp;) : ostream&amp;</td>
</tr>
<tr>
<td>operator &lt;&lt; (double&amp;) : ostream&amp;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>write(char*, streamsize) : ostream&amp;</td>
</tr>
</tbody>
</table>

We have previously used the ______________ operator to write output to a `std::ostream`. The `std::ostream` class itself defines several operator methods to handle primitive types such as `int` and `double`. There are also operator functions to handle the insertion of other types into a `std::ostream`; we have written functions like this. This should all be familiar, but there is one method of `std::ostream` that we have not used yet: the `write` method.
Remember `std::ostream`?

```
string name = /* ... */;
std::cout << name << "\n";
```

```
namespace std {
    ostream& operator << (ostream&, const std::string&);
}
```

Evaluation:

```
(std::cout << name) << "\n";
```

```
(an ostream&) << "\n";
```

Again, as a reminder, when we say that we’re "printing two things with `cout`", what’s actually happening is that we’re evaluating an expression one operation at a time.

For each instance of the insertion operator, the computer calls either an insertion operator `method` or an insertion operator `function`, which actually does the work of printing the value and returns a reference to the same `ostream`. This referenced `ostream` can then be used as the left-hand side argument to another operator method or function, and so on until we’ve completely evaluated the expression.
Remember `std::ostream`?

Can write our own operators:

```cpp
using namespace std;

ostream& operator << (ostream& o, Student& s)
{
    o << s.name() << " (" << s.id() << ");";
    return o;
}
```

We have also written our own insertion operators for `std::ostream` (and it's always good practice to write more!). Whenever we did, we were primarily concerned with outputting text: something that would be readable to a person running our program at the command line.
Simmilarly, we have used the extraction operator with operator methods defined in `std::istream` (and operator functions defined elsewhere) to get information from the user at the console. A simple example follows...
std::istream

```cpp
string name;
unsigned int age;

std::cout << "Please enter your name and age: ";
std::cin >> name >> age;

namespace std {
    istream& operator >> (istream&, const std::string&);
}
(std::cin >> name) >> age;
(an istream&) >> age;
```

This slide shows a simple use of `std::istream` to get a `std::string` and an `int` from the user at the console. Just like the insertion operator and `std::ostream`, extracting values from a stream is a matter of evaluating an expression, one operation at a time.
std::istream

We can write extraction operators too!

```cpp
std::istream& operator >> (std::istream&, StudentID&);
```

We haven’t written a lot of extraction operators so far in the course, but we absolutely can, using the skills we’ve already developed.

*Code example:* Student.h, text-io.cpp, Student.cpp
This is an example of an extraction operator for a `StudentID` structure. First we input an integer like 201612345 from the user, then use integer division and modular arithmetic to split it into a year (since 201612345 / 100000 = 2016) and the serial number (since 201612345 % 100000 = 12345).

*Note:* this only works with mutable objects, as the entire point of the operator is to put data into a `StudentID` (so it can’t be `const`).
Now that we understand inheritance, we can look at a new and very important stream class: `std::fstream`. This class is part of a hierarchy that includes both `std::istream` and `std::ostream`. That is, an `fstream` inherits from `istream` which itself inherits from `ifstream`. Similarly, an `fstream` inherits from `ostream` which itself inherits from `ofstream`.

Because of this inheritance, when we open an `fstream`, we can use any of the methods of `istream` or `ostream` on it, including operators.
Use like any `ifstream` or `ofstream`:

```cpp
ifstream file(filename);
StudentID id;
file >> id;
```

With a couple of caveats:

```cpp
if (!file) { /* ... */ }
StudentID id;
file >> id;
if (file) { /* ... */ }
```

In one way, working with an `fstream` for text-based I/O looks just like interacting with the console via `cin` or `cout`: we use the insertion or extraction operators with any types that have such operators defined.

On the other hand, there are a couple of additional things to consider when performing I/O on files instead of the console. Unlike `std::cout`, which is always connected to the console when any program starts, file streams may not be connected to anything at all if:

1. the file we tried to open doesn’t exist,
2. we don’t have permission to open the file or even
3. we have called the `close()` method on the `fstream`.

To check whether a `std::fstream` is ready for interaction, we can call its `good()` method (or, conversely, its `bad()` method). Also, to check whether or not we have reached the end of a file, we can call the file’s `eof()` method.

*Code example:* `file-io.cpp`
Oftentimes, however, we don’t want to do text-based I/O, because it tends to be quite inefficient. This is partly because we need must more __________ to hold a textual version of a value than its binary equivalent requires and partly because we have to do the work of, e.g., converting an integer into a ones digit, a tens digit, a hundreds digit, etc. (and back again when reading the number!).

The alternative is to use binary I/O: use the `read` and `write` methods of `std::fstream` to get raw bytes out of `/` into a file. We create the `fstream` by passing two values to its constructor: a filename and a set of flags like `ios::out` ("I’m going to read from this file") and `ios::binary` ("I’m going to do binary I/O"). Then, we can call `read` or `write`, passing a `char*` and a length as arguments. C and C++ use `char` arrays to represent raw bytes, since a `char` has a size of one byte.

However, there is a problem: `read` and `write` require a `char*` argument (equivalent to a `char` array), but we don’t have one, we have an object like a `Student` or a structure like a `StudentID`! So... where are we going to get a `char*`?
Binary file I/O

Our Student is an object in memory:

Let us recall that an object in memory, such as a Student object, can be viewed as a grouping of fields stored in memory. In some circumstances (no pointers, no virtual methods, no embedded objects with these things), those fields will be stored contiguously. That is, we could look at that object as a set of contiguous bytes in memory.
Binary file I/O

We could also look at this object as an array of bytes:

What we want to do, in order to use `fstream`'s `read` and `write` methods, is to look at these contiguous bytes explicitly as an array of bytes.
Binary file I/O

"I know this isn't actually a char[], just trust me."

Sounds like a job for...

```
reinterpret_cast
```

Put another way, we want to take an address that points to a value like a StudentID and tell the compiler, "let's treat this as a char array". Of course it isn't a char array, but we're going to ask the compiler to trust us. This sounds a lot like a cast, but not a cast like we've seen before!

Reinterpreting memory, pretending that memory of one type actually contains an entirely unrelated type, requires the use of `reinterpret_cast`. 
Binary file I/O

Recall:

const_cast: treat const value as non-const

static_cast: everything known at compile time

dynamic_cast: only fully known at run time

We have previously seen three casts. const_cast removes a const qualification from a value. static_cast changes a value into another type in a way that can be entirely described statically, at compile time: we can see that some value is a double because that’s it’s type, and we can know at compile time how to convert a double to, e.g., an int.

We’ve also seen dynamic_cast, which relies on run-time or dynamic information: maybe this Shape* values is pointing at a Circle, but then again, maybe it’s actually a Rectangle. We can use dynamic_cast to safely convert from Shape* to Circle* because the cast causes run-time checks to be added (a kind of safety net), and if the Shape isn’t a Circle, we will know about it (because the case will evaluate to nullptr for a pointer conversion or throw a std::bad_cast for a reference conversion).
Binary file I/O

dynamic_cast
  • convert to a subclass reference/pointer
  • dynamic check: "is this correct?" (safety net)

reinterpret_cast
  • convert to any reference/pointer
  • no safety net!

While a **dynamic_cast** incorporates a run-time safety net and will only allow you to downcast from a supertype to a subtype, **reinterpret_cast** will let you convert from any kind of pointer or reference to __________ other kind of pointer or reference. That is, there is no __________.

(incidentally, a "C-style cast", as used in the C programming language, is always a **reinterpret_cast**, just with a shorter syntax that makes it easier to use and easier to use incorrectly)
So, that’s the complete story on binary I/O using `reinterpret_cast` to interpret values as arrays of bytes and write them to or read them from files ...

*Code example: [binary-io.cpp](#)*

... or is it?
Binary file I/O

- a bit simplistic
- assumes complete, contiguous objects
- what about pointers? vtables?

```cpp
class Foo : public Super
{
    Bar* someData;
};
```

The simple kind of binary I/O that we have just studied is fine for values that can be represented as contiguous arrays of bytes, but what if this isn't the case? What if we have an object that contains pointers to other memory allocations? What if we need a vtable to interpret the object? What if we have fields that are objects that themselves contain pointers or vtables?

In these cases, we will need to take a more sophisticated approach. We will need serialization and parsing.
Serialization and parsing

Serialization:
outputting values in a *machine-readable* format

Parsing:
reading values from a machine-readable format

Once we start performing I/O on things that we cannot or wish not to represent as simple blocks of bytes, we need to start doing a bit more work: we need **serialization** and **parsing**.

Serialization allows us to convert values in memory into a *well-defined representation*. This should not depend on the _________________ that generates it (vs, e.g., the way that vtable representations are very C++-specific) or the details of the computer generating the output (e.g., many computers use 4B integers but microcontrollers may use 1B integers). The representation **should** be something that a program can always read in and interpret, again not depending on programming language or computer details.

The process of reading and interpreting data from a file is called **parsing**. This allows us to examine our machine-readable format in a file and reconstitute values in memory.
Recall that we previously used a fairly naïve approach to output. We treated a value in a contiguous block of memory as just an array of bytes, using `reinterpret_cast` to change our interpretation of the memory and pass it to `std::ostream::write()`.

*Code example:* `complex-write.cpp`
Aside

Endianness:

Why is 1 stored as:

```
01 00 00 00
```

When we run the preceding program and inspect its binary output file with a tool like `hexdump` or `xxd -g1 -c8`, we see something rather odd. A number like 1 is not stored as `00 00 00 01` or even as `80 00 00 00`, but as `01 00 00 00`! Why is this? And *why is there an egg on this slide?*

This is a reference to *Gulliver’s Travels*, in particular the people of Lilliput who argue endlessly over whether one should start eating an egg by cracking the big or the little end of an egg. This is emblematic of an arbitrary distinction that people make too much fuss over, and as it turns out, it is the sort of distinction that comes up in representing multi-byte values.
Endianness:

little-endian: least significant byte first

big-endian: most significant byte first

We all agree on where we should find the most significant ________ in a byte, but there is less consensus about where to put the most significant _________ in a multi-byte value (e.g., a 4B integer). There are two popular options: we can write (or transmit) the most significant byte first or the least significant byte first. The first option (MSB first) is known as a ________________ representation; the second (LSB first) is known as ________________:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Big-endian (4B)</th>
<th>Little-endian (4B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x1</td>
<td>00 00 00 01 01 00 00 00</td>
<td></td>
</tr>
<tr>
<td>255</td>
<td>0xff</td>
<td>00 00 00 ff ff 00 00 00</td>
<td></td>
</tr>
<tr>
<td>305,419,896</td>
<td>0x12345678</td>
<td>12 34 56 78 78 56 34 12</td>
<td></td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0x3b9aca00</td>
<td>3b 9a ca 00 00 ca 9a 3b</td>
<td></td>
</tr>
<tr>
<td>4,294,967,295</td>
<td>0xfffffff</td>
<td>ff ff ff ff ff ff ff ff</td>
<td></td>
</tr>
</tbody>
</table>

As it turns out, both of these methods are used in different domains (there is also a middle endian representation, but it’s weird, esoteric and not really used in practice). Wikipedia has a fun description of endianness’ history, but today, most of our desktop and notebook computers use the little-endian convention while most of our networks use big-endian representations. This means that little-endian computers interacting with little-endian computers over a network need to use a big-endian representation for values like their IP addresses and port numbers!
Naive approach:

```c++
const char *raw = reinterpret_cast<char*>(data);
ofstream f;
f.write(raw, sizeof(data));
```

```c++
struct ComplexNumber {
    float real;
    float imaginary;
};
```

This slide reminds of the direct, naïve approach to binary I/O that we saw before: we simply interpret a value in memory as an array of bytes, then write those bytes out to a stream (such as an `ifstream`). This approach, works as long as we are only writing (or reading) values that are contained within contiguous bytes of memory, often called *Plain Old Data (POD)* by C++ programmers. This includes:

1. **primitive types** (e.g., `bool`, `int`, `double`),
2. **arrays** of POD values and/or
3. **structures** containing only POD values.

(technically POD also includes classes with no constructors, destructors or virtual methods, but let's just talk about structures to keep things simple)
More sophisticated approach:

- choose/design a machine-readable format
- serialize into this format
- parse from this format

Once we start dealing with more complex objects that have pointers, virtual methods, etc., we need to explicitly serialize values from memory into a well-defined format and, in the other direction, parse the well-defined format into values in memory. We work with these kinds of well-defined, machine-readable file formats every day: HTML files, GIF/JPEG/PNG images, document files, zip files, etc. Before we can serialize to a format or parse from it, however, we need to design a format (if an appropriate one already exists) or else design a new format.
Example from Student.h:

1. a signature: the string "STU1"
2. name length (8 B)
3. name
4. student ID
   a. year (4 B)
   b. serial number (4 B)

**Question:** what's the point of the signature?

**Answer:**

This kind of "signature" (also known as a magic number) helps us check whether or not we've got the right sort of file. If we are reading, e.g., a PNG image file, the first thing to do is check whether or not the file starts with the bytes 89 50 4e 47 0d 0a 1a 0a, which include the letters "PNG". In this example, I've chosen the string "STU1" to indicate that this is version 1 of a format for representing Student objects.

Next, to represent the first name in the Student object (std::string name_), we need to write out all of the character values inside the string. If that's all we did, however, we would have a problem when parsing: how would we know when to stop interpreting bytes from the file as the name? In order to make things clear and unambiguous, we should first write the ____________ of characters, followed by the characters themselves.

Finally, to write the StudentID field, we can use a little format embedded within our larger format. Here, because we are using 4B integers, we can simply write them out as primitive values: arrays of bytes. We ought to also specify which endianness we're using in this format to ensure that there is no confusion.
Data formats:

- write them down!
- check the data!
  - signatures
  - reasonable values

There are a few simple rules that you should follow when working with file formats to prevent problems. First, _________________. It doesn’t have to be a complicated standard ratified by ISO member countries, but a written specification helps clarify exactly what you mean for your format to express.

Second, when you are working with data that’s been encoded in a particular format, you should guard against accidents (e.g., opening the wrong file) and corruption by _________________. If the format has a signature, check that your data contains this signature! If you’re reading values that have well-defined semantics (e.g., year that a student entered Memorial University), check that those values make sense (e.g., not in the future, not before the University opened).
As an example, we will work through the process of writing these serialization and parsing methods in class on Friday.
Summary

File I/O

Serialization

Parsing
Topic 20:

Higher-order functions
This slide shows a simple example of a function that takes double-precision floating-point number (in this case, an angle) and converts it into another double-precision floating-point number (in this case, the sine of the angle). We can do three things with functions: we can ____________ them, ____________ them and _____________. This should all look very, very familiar!
Graphing library:

```c++
struct Point { double x; double y; };
void drawGraph(const std::vector<Point>& points);
```

Usage:

```c++
vector<Point> points;
for (double x = -M_PI; x <= M_PI; x += 0.01)
{
    points.push_back({ x, sin(x) });
}
drawGraph(points);
```

Now, suppose that we had a graphing library that we wanted to use to plot the values of \( \sin(x) \) for various values of \( x \) on an X-Y scatter plot. Suppose this graphing library has a function that takes a `vector<Point>` as a parameter: we need to construct a vector of `Point` structure instances and then call `drawGraph()`, at which point a graph will appear on the screen.

One way we might use this library is as shown in the second code example. In this example, we calculate the value of \( \sin(x) \) for many values of \( x \) in the range \([−\pi, \pi]\). For each of these calculations, we add a `Point` containing \( x \) and \( \sin(x) \) to a `vector<Point>` and then pass it to `drawGraph()`.
Another function:

\[ \text{double cos(double);} \]

Usage:

\[
\begin{align*}
\text{vector<Point> points; for (double x = -M_PI; x <= M_PI; x += 0.01)} & \\
& \{ \\
& \quad \text{points.push_back(\{ x, \cos(x) \});} \\
& \} \\
& \text{drawGraph(points);} \\
\end{align*}
\]

This slide shows another example in which we are graphing a trigonometric function, but this time we're graphing the cosine instead of the sine function. Once again, we calculate the value of \( \sin(x) \) for many values of \( x \) in the range \( [-\pi, \pi] \). This time, instead of creating a Point containing \( x \) and \( \sin(x) \), we create a Point containing \( x \) and \( \cos(x) \). Just like before, however, we add all of these points to our \text{vector<Point>} \ and pass it to \text{drawGraph()}.
Compare:

```cpp
vector<Point> points;
for (double x = -M_PI; x <= M_PI; x += 0.01)
{
    points.push_back({ x, sin(x) });
}
drawGraph(points);
```

```cpp
vector<Point> points;
for (double x = -M_PI; x <= M_PI; x += 0.01)
{
    points.push_back({ x, cos(x) });
}
drawGraph(points);
```

If we compare these two implementations, we'll see an awful lot of commonality:

- both declare a `vector<Point>`
- both have a `for` loop that iterates through lots of trig-ranged values of `x`
- both create a `Point` for each such value and add it to the vector
- both pass the resulting vector to `drawGraph`
Nearly identical!

```
for (double x = -M_PI; x <= M_PI; x += 0.01)
{
    points.push_back({ x, sin(x) });
+    points.push_back({ x, cos(x) });
}
```

graph(points);

**Imagine the same for** \(\tan, \sec, \csc, \log, \text{etc...} \)**

**need an abstraction for**

"a function that converts a double to a double"

In fact, the *only* difference between these two code examples is ___________ that they call! Now, programmers *hate* code ___________: not only does it require useless work, it adds risk. If we copied and pasted our \(\sin\) code once and then modified it to use \(\cos\), there's a reasonable chance we'd get it right, but what happens when we want to ___________ the code to use some new API? We will have to *remember* to change both versions!

This problem gets much worse if we don't just have two copies of the code but three, four, five or more. Every time we make another copy of essentially the same code, we are increasing the risk that we might mess it up and we are *also* increasing the maintenance ___________ of our code base.

When we encountered duplication before, we found ways to replace it with some new ___________: a class that we can create multiple objects of, a superclass that we create multiple subclasses of, or even (hearkening back to purely ___________ programming), a function that we call multiple times with different arguments used to initialize its parameters. But what do we do when the thing we'd like to change every time we run a function (the ____________) is itself a function?
This abstraction for a function as something with certain parameters and return type is provided for us in C++11 and up in the `std::function` type. Objects of type `std::function` represent a function that can be used with arguments of a specified parameter types in order to yield a value of a specified type.

Function declarations, as we have seen them so far, include a type (not a "function type"), a name and a list of (which do not have to be named). The `std::function` type conveys all of the same information about what a function takes as its parameters and what it returns, but ignores the.

Examples of `std::function` use on this slide show how we can create an object to represent either the `sin` function or the `cos` function. Both have the same parameter types and return type, so we can represent either of them as "a function that takes a double and returns a double" (which we call the function type, as distinguished from just the by itself).

Code example: `trig.cpp`
Higher-order functions

Graphing example:

```cpp
void graph(std::function<double (double)> f)
{
    vector<Point> points;
    for (double x = -M_PI; x <= M_PI; x += 0.01)
    {
        points.push_back({ x, f(x) });
    }
    graph(points);
}

graph(sin);
graph(cos);
```

This slide takes us back to our example of trying to graph a trigonometric function. Now, instead of having two (or six!) blocks of nearly-identical code to graph the different trig functions, we can have one function for graphing "a trig function", and we can ___________ that function and pass it different trigonometric functions as ___________. These kinds functions that treat functions as values rather than named functions are called higher-order functions.
Using functions as values:

```cpp
std::function<double(double)> f = /* ... */;

void graph(std::function<double(double)> f)
{
    /* ... */
}

std::function<double(double)> trigFunction(std::string name)
{
    if (name == "sin")
        return sin;
    /* ... */
}
```

Higher-order functions

There are three ways to use functions as values:

1. we can directly assign functions to `std::function` objects (or, in older C implementations, to `function pointers`),

2. we can pass function **arguments** into a function’s **parameters** and/or

3. we can return functions from functions.
Another example:

```cpp
bool lessThan(double x, double y) { return (x < y); }
bool greaterThan(double x, double y) { return (x > y); }

typedef std::function<bool(double, double)> Comparator;

void sort(std::vector<double>&, Comparator);

sort(values, lessThan);
sort(values, greaterThan);
```

This kind of abstraction and generalization over functions is a very useful thing to do, and C++11 and later make it pretty easy. This slide shows an example of how we could implement a function (or a method) for sorting data without having to know what the sort order is. That way, we can use the same sorting code whether we’re trying to sort from least to greatest, A to Z, Z to A, according to final grade, etc.
Summary

Functions

Higher-order functions
Returning a function:

```c++
double sin(double);
double cos(double);
double identity(double x) { return x; }
double triple(double x) { return 3 * x; }
```

```c++
std::function<double (double)> getFunction()
{
    return triple;
}
```

This slide reminds us of how we can return a function from a function (an example of higher-order functions). We are returning a `std::function<double (double)>`, which represents a function that takes a `double` parameter and returns a `double` value. In this case, we are returning the `triple` function, which takes a value \( x \) and returns \( 3x \).
Returning functions

How could this be useful?

```cpp
typedef std::function<double (const std::vector<Student>&)> StatFn;

StatFn f = letTheUserChooseAStatisticsFunction();

vector<Student> students = Student::ParseStudents(file);
double statistic = f(students);
```

We use higher-order functions to decouple the selection of which function to use from the actual function call. In this example, we can have one bit of code that’s responsible for asking the user, "which statistic would you like to compute?" and another bit that’s responsible for loading student data and passing it to the number-crunching code.

This kind of division of responsibility is good software design, as it allows a lot of flexibility in our implementation: the code for interacting with the user can be completely separate from the number-crunching code and the data-parsing code, written by different people, tested independently and resilient to changes in details of how the other code works.
Returning functions

How could this be useful?

```cpp
double degrees(double rad) { return 180 * rad / M_PI; }
double radians(double deg) { return M_PI * deg / 180; }
double kilometers(double miles) { return 1.60934 * miles; }
double miles(double km) { return km / 1.60934; }
```

```cpp
std::function<double (double)> multiplier(double);

function<double (double)> convert = multiplier(M_PI / 180);

double rad = convert(90);
```

Now suppose that we wanted to work with code that needs to apply a scalar conversion factor to some numbers. This could be a conversion between degrees and radians, miles and kilometers or any linear scalar conversion. This can be generalized to the idea of a ____________ function: something that takes any given value and multiplies it by a fixed value.

It could be convenient, in many applications, to be able to construct an arbitrary conversion function. We know how to do this with existing functions like `sin` or `cos`, but how can we create a conversion function that will take any scalar value as a parameter?
Returning functions

How can we implement this?

- another way: lambda function

```cpp
Function<
    double (double)
> multiplier(double m)
{
    return [m](double x) { return m * x; };
}
```

- function created "on the fly"

  no names: just a std::function value

This is what a __________ is: a function created "on the fly" as a __________ rather than using a traditional function name. We can then use the function as a std::function object, passing it around to parameters, variables and as return values, all without ever giving the function a __________.
Lambda functions

\[
[m](\text{double } x) \{ \text{return } m \ast x; \}
\]

Capture specifier:

\[ [m] \]

Parameters:

\[(\text{double } x) \]

Body:

\[
\{ \text{return } m \ast x; \}
\]

A lambda function has three crucial syntactic elements, two of which look a lot like a regular function. These are a ________________, some ______________ and a ____________.

**Question:** which of these elements is not found in a "normal" function?

______________________________
Lambda functions

Capture specifier:

```c
int multiplier = /* ... */;
return [multiplier](double x) { return multiplier * x; }
```

- how to "capture" surrounding values
- like parameters passed when the lambda is created rather than executed

The capture specifier allows us to inject values into the lambda function that are parameterized when the function is defined rather than when it is ______________. "Injecting" a value into the lambda function like this is properly called capturing the value.
### Lambda functions

**Capture specifier:**

```c
int multiplier = /* ... */;
return [multiplier](double x) { return multiplier * x; }
```

- by value: `[m]() { /* ... */ }`
- by reference: `([&m]() { /* ... */ })`
- default by value: `[]=() { /* ... */ }`
- default by reference: `[&]() { /* ... */ }`

There are four ways that we can specify that a value be captured:

1. explicitly, by value
   - if a variable is explicitly mentioned by name in the capture specified, its value is **copied** into the lambda function (which may invoke a **copy constructor** if the value is an object)

2. explicitly, by reference
   - if a variable is mentioned by name with an ampersand, it is not copied into the lambda function — instead, the lambda function will have a **reference** to the variable

3. implicitly, by value
   - if the `=` sign is present in a capture specifier, the lambda function will automatically capture whatever additional values it needs from the surrounding scope by value

4. implicitly, by reference
   - if the `=` sign is present by itself in a capture specifier (i.e., not next to a variable name), the lambda function will automatically capture whatever additional values it needs from the surrounding scope by **reference**
Capture specifier:

```cpp
t int multiplier = /* ... */;
return [&multiplier](double x) { return multiplier * x; }
```

- be careful with references to stack
- can have dangling references!

It is important to be careful about references in the capture specifier. If the lambda function captures a variable by reference, but the lambda function gets called after the value it referenced is deallocated (e.g., a local variable goes out of scope), we can end up with a dangling reference. This is very bad, for all of the reasons that we talked about when we looked at dangling pointers.

Unless you have a specific reason to capture by reference, consider capturing by ______ as your first instinct.
Parameters:

```cpp
int multiplier = /* ... */;
return [multiplier](double x) { return multiplier * x; }
```

- just like parameters to normal functions
- require arguments to be passed in call

Parameters to lambda functions operate in the same way as ordinary function parameters. When any function is called, including a lambda function, arguments are passed as part of the call to initialize the parameters inside the function. Those parameters can then be used as local variables inside the function.
Body:

```c
int multiplier = /* ... */;
return [multiplier](double x) { return multiplier * x; }
```

- just like body of normal functions
- a series of statements

The body of a lambda function is also just like a normal function: a block of statements that are executed, one at a time, until a `return` statement or the end of the function.