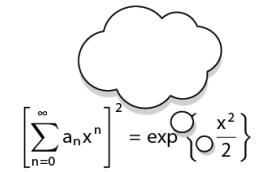
Why C++?

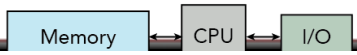
- General-purpose
- Multi-paradigm(ish)
- Efficient
- Employable
- Foundational




```
instance  Functor Maybe  where
    fmap _ Nothing       = Nothing
    fmap f (Just a)      = Just (f a)

instance  Monad Maybe  where
    (Just x) >>= k       = k x
    Nothing  >>= _       = Nothing
```

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

```
zstr_count:         ; Entry point
    mov  ecx, -1    ; Init the loop counter
.loop:
    inc  ecx        ; Add 1 to the loop counter
```



_____: 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++. _____ _____, and understanding C++ will help you _____

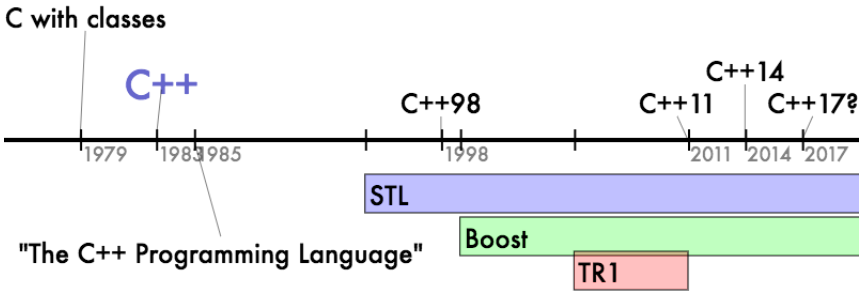
_____: thanks to C++'s flexibility, you can write procedural, object-oriented, functional (more and more) or just about any other kind of program you might want.

_____: the implementers of C++ are relentless about it being _____. 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 _____ 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 _____.
Understanding C++ gets you down the road to all of these.

History of C++


C with classes
C++
C++98
C++11
C++14
C++17?
1979
1983
1985
1998
2011
2014
2017
STL
Boost
TR1
"The C++ Programming Language"

C++11

- Significant language improvements
- Safety features
- Simpler *and* more complex!
- Legacy

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

Types

```
int i = 42;
double x = 3.1415926;
string s = "Hello, world!";
```

Questions:

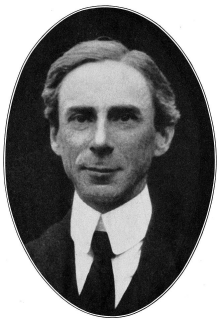
- what do these types tell us?
- how have you used these types?
- when were types created?

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 _____ _____.

History of types

Set theory:

$$S = \left\{ x | \frac{x}{2} \in \mathbb{Z} \right\}$$



Russell's paradox:

$$\text{Let } R = \{x | x \notin x\}$$

$$\text{then } R \in R \iff R \notin R$$

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*.

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

$$S = \left\{ x | \frac{x}{2} \in \mathbb{Z} \right\}$$

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.

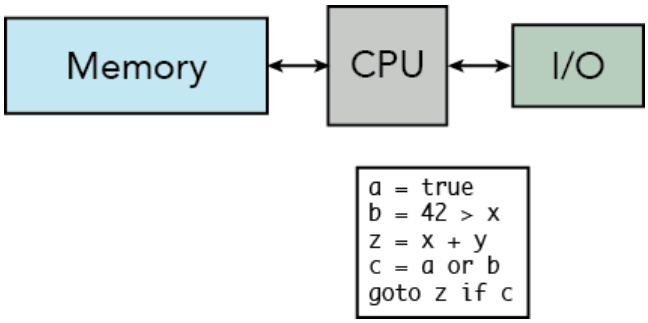
In the *typed* set theory, we add a constraint to S: we specify that S is a _____.

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.

Types in computing

Abstract machine:



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 _____?

Types in computing

Memory

`0100 0000 0100 1001 0000 1111 1101 1010`

32b `int` : 1,078,530,010

32b `float` : 3.1415926

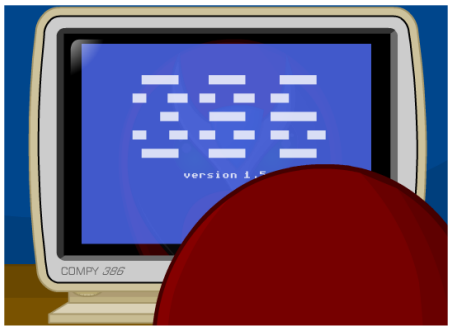
Here is a depiction of some memory. Memory contains 0's and 1's. But what do they mean?

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 _____.

Types in processors




```
mul       ; unsigned integer multiplication
imul      ; signed integer multiplication
fmul      ; floating-point multiplication
mulps     ; multiply packed single-precision floats
vfmaddpd  ; fused multiply-add of packed double-pre...
; even more!
```


To get even more practical, let's look at a couple of examples from a more realistic processor (not an abstract machine).

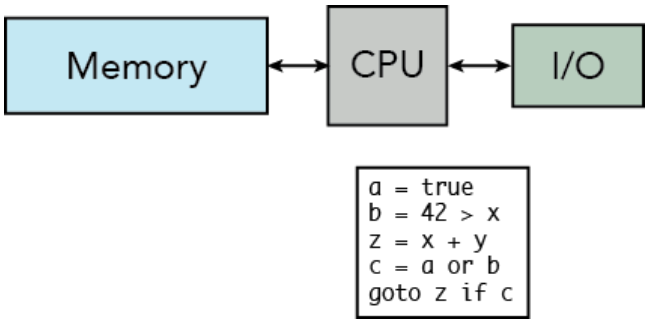
All of your notebook computers, desktop computers, etc., have processors that use some variant of the x86 instruction set (named for the 8086, 286, 386, 486, etc.). The instruction set is the _____ _____ with a processor: a program (such as a C++ program) is translated by the compiler into a _____ taken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

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.

Types in programming languages

Abstract machine:



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.

Types in C++

Basic types:

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

C and C++ have a number of _____ 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:

Type	Size
`bool`	1 B
`char`	1 B
`int`	4 B
`float`	4 B
`double`	8 B

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

Types in C++

Specific integers

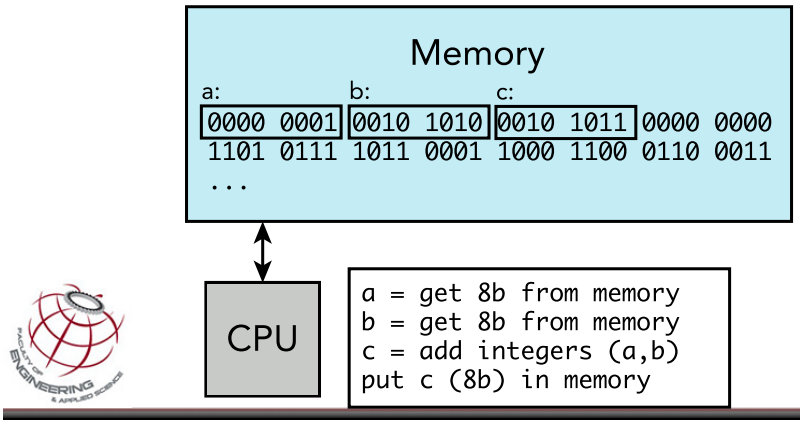
```c
#include <stdint.h>

// 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).

Types in C++

```
int8_t a = /* ... */;
int8_t b = /* ... */;
int8_t c = a + b;
```



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

```
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

Representation

Operations

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.

Operations

Common Operators:

Operator	Operation
==	Equals
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

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 equalty operator is used to _____ for equality between two expressions. When we want to _____ two things equal, we need to use the *assignment operator* (=).

Operations

Common Operators

Operator	Operation	Applies to
`+`	Add	`int`, `double`
	Concatenate	`string`
`-`	Subtract	`int`, `double`
`*`	Multiply	`int`, `double`
`/`	Divide	`int`, `double`
`%`	Modulus	`int`

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

Operator	Operation	Applies to
+=	Add & assign	`int`, `double`
	Append	`string`
-=	Subtract & assign	`int`, `double`
*=	Multiply & assign	`int`, `double`
++	Increment	`int`, `double`, iterators, ...
--	Decrement	`int`, `double`, iterators, ...

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.

Exercise

Given:

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

Try:

```
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.