Topic 3:

Expressions

Mathematical expressions

(Very) rough definition:

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

$$4$$

$$\nabla \times \vec{E}$$

$$-\frac{\partial \vec{B}}{\partial t}$$

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}$$

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 ($\nabla \times \vec{E}$) is a three-dimensional vector with units T/s (teslas per second).

C++ expressions

Numbers:

```
4
```

```
0.5 * sin(3.1415926)
```

Memory and objects:

```
some_array[4]
```

```
string("Hello, world!")
```

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`).

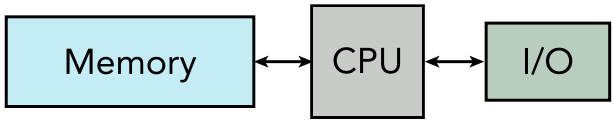
You can explore some of these expressions by modifying the expression-types.cpp code example.

What just happened?

"Simple" addition:

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

Not so simple!



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


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`!

Addition

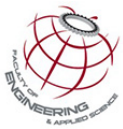
Machine can add:

```
int + int
double + double
```

How can we add:

```
int + double
```

?



Given that our abstract machine can add an `int` to an `int` or a `double` to a `double`, we need to do something else if we're going to be able to add an `int` to a `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 `int + 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

Numeric promotion

Safe!

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

```
// 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 0000 0110`). This is demonstrated in the numeric-promotion.cpp _____ _____.

Numeric promotion

Floating-point too:

```
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.

Numeric conversion

Unsafe conversions also allowed!

```
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 signedVersion = largeInteger;
            ~~~~~~~~~~~~~   ^~~~~~~~~~~~
unsafe-conversions.cpp:13:12: warning: implicit conversion loses floating-point precision: 'doubl
        float f = d;
              ~   ^
unsafe-conversions.cpp:20:18: warning: implicit conversion turns floating-point number into integ
        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.

Numeric conversion

Floating-point less shocking:

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

Unless it's not:

```
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.

Boolean tests

Safe(ish)

```
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