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.

static_cast

Do something "fairly safe":

```
static_cast<IntendedType>(expression)
```


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

Mathematical equations

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

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

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t}$$

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

A mathematical equation is simply a declaration that ____ ________________. 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.

C++ assignment

```
bool b = true;
short s = 1000;
double x = 3.1415926;
string hi = "hello, world!";
```

More than just math:

- Name
- Space

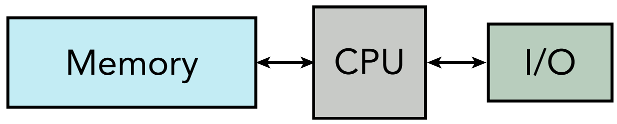
Assignment in programming is about putting a value into a ______________. This is different from mathematical equality ($2x = y$), even though it uses the same equals sign. Instead, assignment is more like the definition of a mathematical variable $\left(\text{let } x = \frac{y}{2}\right)$.

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 $\text{let } x = 3.1415926$.

Unlike mathematical variables, however, variables in programming languages are also associated with ____________.

Variables

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:

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

Initialization:

```
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.

Variables

Declaration:

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

    // ...
}
```

Memory

```
b   0101 0101

s   0101 0101
    0101 0101

x   0101 0101
    0101 0101
    0101 0101
    0101 0101
    0101 0101
    0101 0101
    0101 0101
    0101 0101
```


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.

Variables

Initialization:

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

Memory	
b	0000 0001
s	0000 0011
	1110 1000
x	0100 0000
	0000 1001
	0010 0001
	1111 1011
	0100 1101
	0001 0010
	1101 1000
	0100 1010

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.

L- and R-values

```
b = true;
s = 1000;
```

Expressions:

```
b
```

```
true
```

```
s
```

```
1000
```


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?

L- and R-values

What's the difference?

```
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???

```
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`) __________ 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

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

L or R?

```
c
```

```
42
```

```
{ 4.1, 3.9, 2.4 }
```

```
forces[2]
```


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 `double`s 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.

References

Previously:

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

More generally:

```
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

References:

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

```
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- and R-values

L-values have memory

R-values do not

References are L-values that share