Operators in C++

There are four kinds of operator in C++. Classifying operators according to their kind makes it easy to see how to use them.

1. Dyadic.
   (This is very frequently called Binary instead, but that is confusing because everything on a computer is binary, the word is being used in a slightly different sense).
   Dyadic operators are the most familiar kind, they appear between the two values, `*` for multiplication is an example. To multiply 6 by nine you write `6 * 9`.

2. Unary Prefix
   (Sometimes people say Monadic instead of Unary, but that is quite unusual).
   They are also very familiar, they only work on one value, and are placed before it. As well as its very familiar use as a dyadic operator, – can also be a unary prefix operator. If you want a negative seven (or minus seven as I prefer to call it), just put a – sign in front of it, the same as in mathematics: `– 7`.

3. Unary Postfix
   These are like the fairly familiar use of `!` for factorials. They also only work on a single value, but they are written after it. C++ does not use `!` for factorial, it means something else. None of the postfix operators work on plain numbers, so I'll have to expand our domain a bit. If you have stored a number in a variable named `x` and you want to add one to it, one way is to follow it by the `++` operator: `x ++`. We haven't talked about variables yet.

   Function calls are also considered to be unary postfix operations. Although you will typically have a number of values involved in a function call, like `move_to(120, 30)`, the bracket pair `()` are seen as an operator being applied to the function `move_to`.

4. Conditional Expression
   (some people say Triadic or Ternary, but there is only one such operator, so it seems a little pretentious to use a fancy generalisation. Using its name sounds better)
   Three things are involved in a conditional expression, and it looks like this: `6 < 7 ? 4 : 8`. The `?` and `:` are considered to be parts of the same operator. It looks as though four things are involved in that example, but the `? :` is only dealing with three things: `6 < 7` and `4` and `8`. The value of that expression is `4` if `6 < 7` turns out to be true, and `8` if it doesn't.

Operators also have priorities, a lower number represents a higher priority. This too is perfectly familiar. Everybody knows that `2 + 3 × 4` is supposed to be `14`, not `20`.

I will list the operators that you will see in this class first. Then I'll give the whole list for future reference.

Priority	Kind	General use	operator	meaning
1	dyadic	namespaces	`::`	distinguishing names
2	unary post	varied	`++`	increment
			`--`	decrement
			`()`	function call
			`[]`	array access
			`.`	access member
3	unary pre	arithmetical	`++`	increment
			`--`	decrement
			`+`	does nothing
			`-`	negate
		logical	`!`	swap true and false
			`~`	swap 0 and 1 in binary values
		typecast	`()`	e.g. `(int)3.14159`
5	dyadic	arithmetical	`*`	multiplication
			`/`	division
			`%`	modulo, or find remainder
6	dyadic	arithmetical	`+`	addition
			`-`	subtraction
7	dyadic	binary shift	`<<`	shift left
			`>>`	shift right
8	dyadic	comparison	`<`	less than
			`>`	greater than
			`<=`	less than or equal
			`>=`	greater than or equal
9	dyadic	comparison	`==`	equality
			`!=`	inequality
10	dyadic	logical	`&`	and on binary digits
11	dyadic	logical	`^`	exclusive or on binary digits
12	dyadic	logical	`|`	or on binary digits
13	dyadic	logical	`&&`	and on true or false
14	dyadic	logical	`||`	or on true or false
15	dyadic	update	`=`	assignment
			`*=`	multiply and update
			`/=`	divide and update
			`%=`	modulo and update
			`+=`	add and update
			`-=`	subtract and update
			`&=`	binary and and update
			`|=`	binary or and update
			`^=`	binary exclusive or and update
			`<<=`	left shift and update
			`>>=`	right shift and update
	triadic	conditional	`? :`	if A then B otherwise C
	dyadic	ordering	`,`	calculate and forget

There is one further distinction to be made. If a sequence of operators have the same priority, as in `2 * 3 / 4`, they happen from left to right. The one exception is for all of the operators with priority 15. `a = b = c` is the same as `a = (b = c)`, and `a ? b : c ? d : e` is the same as `a ? b : (c ? d : e)`.

Some operators have extra meanings for different types. For example, the dyadic `+` doesn't only add numbers together, it can be used on strings too, taking `"hipp"` and `"opotamusses"` and producing `"hippopotamusses"`.

Generally, the result an operator produces has the same type as the values it was given. In `8 / 3`, the two operands are both whole numbers, `int`s, so the result will also be an `int`. Everything after where the decimal point would be is thrown away. The value of `8 / 3` is `2`. If you want the "real" answer, say `8.0 / 3.0` instead.

In numeric operations, sometimes the values an operator gets have different types. In that case, there is a very simple rule. The value that has the least precision is converted to have the same type as the other. `double`s have more precision than `float`s, and `float`s have more precision than `int`s. (but remember, there is hardly ever any reason to use `float` any more). So if the expression is `8 / 3.0`, the `8` is promoted to `8.0` and the answer is `2.666666`...

Later on, you will come across different sizes of `int`s. A normal `int` almost always has a range of approximately ±2,000,000,000, but there is also a smaller version with a range of about ±33,000, and a larger version with an approximate range of ±9,000,000,000,000,000,000. The rule works the same way, a value with a smaller range is converted to the type with the larger range.

In all cases, `float`s and `double`s are considered to be more precise or bigger than any kind of `int`.

This is the whole list of operators, including those you won't meet until another semester.

Priority	Kind	General use	operator	meaning
1	dyadic	namespaces	`::`	distinguishing names
2	unary post	varied	`++`	increment
			`--`	decrement
			`()`	function call
			`[]`	array access
			`.`	access member
			`->`	access member through pointer
3	unary pre	arithmetical	`++`	increment
			`--`	decrement
			`+`	does nothing
			`-`	negate

		logical	`!`	swap true and false
			`~`	swap 0 and 1 in binary values
		pointers	`&`	create a pointer
			`*`	follow a pointer
		typecast	`()`	e.g. `(int)3.14159`
		measuring	`sizeof`	how many bytes something fills
		memory allocation	`new`	request more memory
			`delete`	give unneeded memory back
4	dyadic	member ptr	`.*`	general access to members
			`->*`	as above but through pointer
5	dyadic	arithmetical	`*`	multiplication
			`/`	division
			`%`	modulo, or find remainder
6	dyadic	arithmetical	`+`	addition
			`-`	subtraction
7	dyadic	binary shift	`<<`	shift left
			`>>`	shift right
8	dyadic	comparison	`<`	less than
			`>`	greater than
			`<=`	less than or equal
			`>=`	greater than or equal
9	dyadic	comparison	`==`	equality
			`!=`	inequality
10	dyadic	logical	`&`	and on binary digits
11	dyadic	logical	`^`	exclusive or on binary digits
12	dyadic	logical	`|`	or on binary digits
13	dyadic	logical	`&&`	and on true or false
14	dyadic	logical	`||`	or on true or false
15	dyadic	update	`=`	assignment
			`*=`	multiply and update
			`/=`	divide and update
			`%=`	modulo and update
			`+=`	add and update
			`-=`	subtract and update
			`&=`	binary and and update
			`|=`	binary or and update
			`^=`	binary exclusive or and update
			`<<=`	left shift and update
			`>>=`	right shift and update
	triadic	conditional	`? :`	if A then B otherwise C
	dyadic	ordering	`,`	calculate and forget