3.4 — Sizeof, comma, and conditional operators

BY ALEX ON JUNE 14TH, 2007 | LAST MODIFIED BY ALEX ON APRIL 1ST, 2018

Sizeof operator

Operator	Symbol	Form	Operation

Sizeof	sizeof	sizeof(type)
		sizeof(variable)
			Returns size of type or variable in bytes

We covered the sizeof(type) form of the operator in lesson **2.3 -- Variable sizes and the sizeof operator**. Refer to that lesson for more information on the sizeof(type) form of the operator.

Note that sizeof can also be used on a variable, and will return the size of that variable:

```
#include <iostream>

int main()
{
    double d = 5.0;
    std::cout << sizeof(d); // will print the size of variable d in bytes
}
```

Comma operator

Operator	Symbol	Form	Operation
Comma	,	x, y	Evaluate x then y, returns value of y

The comma operator allows you to evaluate multiple expressions wherever a single expression is allowed. The comma operator evaluates to its rightmost operand.

For example:

```
int x = 0;
int y = 2;
int z = (++x, ++y); // increment x and y
```

z would be assigned the result of evaluating ++y, which equals 3.

In almost every case, a statement written using a comma would be better written as separate statements. For example, the above code should be written as:

```
int x = 0;
int y = 2;
++x;
++y;
int z = y;
```

Note that comma has the lowest precedence of all the operators, even lower than assignment. Because of this, the following two lines of code do different things:

```
z = (a, b); // evaluate (a, b) first to get result of b, then assign that value to variable z.
z = a, b; // evaluates as "(z = a), b", so z gets assigned the value of a, and b is discarded.
```

Most programmers do not use the comma operator at all, with the single exception of inside *for loops*, where its use is fairly common. We discuss for loops in a future lesson.

Note that although a comma is used to separate the parameters in a function call, this is not using the comma operator.

```
int sum = add(x, y); // this comma is not the comma operator
```

Similarly, when declaring multiple variables on a single line, the comma there is just a normal separator, not the comma operator

```
int x(3), y(5); // this comma is not the comma operator either
```

Rule: Avoid using the comma operator, except within for loops.

Conditional operator

Operator	Symbol	Form	Operation
Conditional	?:	c ? x : y	If c is nonzero (true) then evaluate x, otherwise evaluate y

The conditional operator (?:) (also known as the "arithmetic if" operator) is C++'s only ternary operator (it takes 3 operands). Because of this, it's also sometimes referred to as the "ternary operator" (we suggest you avoid calling it this in case C++ adds another ternary operator in the future).

The ?: operator provides a shorthand method for doing a particular type of if/else statement.

If/else statements in the following form:

```
if (condition)
    expression;
else
    other_expression;
```

can be rewritten as:

```
(condition) ? expression : other_expression;
```

Note that the operands of the conditional operator must be expressions themselves (not statements).

For example, an if/else statement that looks like this:

```
if (condition)
    x = some_value;
else
    x = some_other_value;
```

can be rewritten as:

```
x = (condition) ? some_value : some_other_value;
```

Many people find this more compact form easier to read.

As another example, to put the larger of values x and y in variable larger, we could write this:

```
if (x > y)
    larger = x;
else
    larger = y;
```

Or this:

```
larger = (x > y) ? x : y;
```

It is common to put the conditional part of the expression inside of parenthesis, both to make it easier to read, and also to make sure the precedence is correct.

It is also worth noting that the expression between the ? and : is evaluated as if it were parenthesized.

Keep in mind that the ?: operator has a very low precedence. If doing anything other than assigning the result to a variable, the ?: statement needs to be wrapped in parenthesis.

For example to print the larger of values x and y to the screen, we could do this:

```
if (x > y)
    std::cout << x;
else
    std::cout << y;
```

Or we could do this:

```
std::cout << ((x > y) ? x : y);
```

Because the << operator has higher precedence than the ?: operator, the statement:

```
std::cout << (x > y) ? x : y;
```

would evaluate as:

```
(std::cout << (x > y)) ? x : y;
```

That would print 1 (true) if x > y, or 0 (false) otherwise!

The conditional operator gives us a convenient way to simplify simple if/else statements, particularly when assigning the result to a variable or returning the result as part of a function return. It should not be used for complex if/else statements, as it quickly becomes both unreadable and error prone.

Rule: Only use the conditional operator for simple conditionals where it enhances readability.

The conditional operator evaluates as an expression

It's worth noting that the conditional operator evaluates as an expression, whereas if/else evaluates as a set of statements. This means the conditional operator can be used in some places where if/else can not.

For example, when initializing a const variable:

```
#include <iostream>

int main()
{
    bool inBigClassroom = false;
    const int classSize = inBigClassroom ? 30 : 20;
    std::cout << "The class size is: " << classSize;

    return 0;
}
```

There's no satisfactory if/else statement for this. You might think to try something like this:

```
#include <iostream>

int main()
{
    bool inBigClassroom = false;
    if (inBigClassroom)
        const int classSize = 30;
    else
        const int classSize = 20;
    std::cout << "The class size is: " << classSize;

    return 0;
}
```

However, this won't compile, and you'll get an error message that classSize isn't defined. Much like how variables defined inside functions die at the end of the function, variables defined inside an if or else statement die at the end of the if or else statement. Thus, classSize has already been destroyed by the time we try to print it.

If you want to use an if/else, you'd have to do something like this:

```
#include <iostream>

int getClassSize(bool inBigClassroom)
```

```
{
    if (inBigClassroom)
        return 30;
    else
        return 20;
}

int main()
{
    const int classSize = getClassSize(false);
    std::cout << "The class size is: " << classSize;

    return 0;
}
```

This one works because we’re not defining variables inside the if or else, we’re just returning a value back to the caller, which can then be used as the initializer.

That’s a lot of extra work!

The type of the expressions must match or be convertible

To properly comply with C++’s type checking, both expressions in a conditional statement must either match, or the second expression must be convertible to the type of the first expression.

So while you might expect to be able to do something like this:

```
#include <iostream>

int main()
{
    int x = 5;
    std::cout << (x != 5 ? x : "x is 5"); // won't compile

    return 0;
}
```

The above example won’t compile. One of the expressions is an integer, and the other is a string literal. The compiler will try to find a way to convert the string literal to an integer, but since it doesn’t know how, it will give an error. In such cases, you’ll have to use an if/else.

70 comments to 3.4 — Sizeof, comma, and conditional operators