5.5 — Comma and conditional operators

ALEX AUGUST 6, 2022

The 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 the left operand, then the right operand, and then returns the result of the right operand.

For example:

```
#include <iostream>

int main()
{
    int x{ 1 };
    int y{ 2 };

    std::cout << (++x, ++y) << '\n'; // increment x and y, evaluates to the right operand

    return 0;
}
```

First the left operand of the comma operator is evaluated, which increments x from 1 to 2. Next, the right operand is evaluated, which increments y from 2 to 3. The comma operator returns the result of the right operand (3), which is subsequently printed to the console.

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 evaluated and discarded.
```

This makes the comma operator somewhat dangerous to use.

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

```
#include <iostream>

int main()
{
    int x{ 1 };
    int y{ 2 };

    ++x;
    std::cout << ++y << '\n';

    return 0;
}
```

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 future lesson 7.9 -- For statements (https://www.learncpp.com/cpp-tutorial/for-statements/).

Best practice

Avoid using the comma operator, except within for loops.

Comma as a separator

In C++, the comma symbol is often used as a separator, and these uses do not invoke the comma operator. Some examples of separator commas:

```
void foo(int x, int y) // Comma used to separate parameters in function definition
{
    add(x, y); // Comma used to separate arguments in function call
    constexpr int z{ 3 }, w{ 5 }; // Comma used to separate multiple variables being defined on the same line (don't do this)
}
```

There is no need to avoid separator commas (except when declaring multiple variables, which you should not do).

The 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 sometimes called the "arithmetic if" operator) is a ternary operator (it takes 3 operands). Because it has historically been C++'s only ternary operator, it's also sometimes referred to as "the ternary operator".

The ?: operator provides a shorthand method for doing a particular type of if/else statement. Please review lesson 4.10 -- Introduction to if statements (https://www.learncpp.com/cpp-tutorial/introduction-to-if-statements/) if you need a brush up on if/else before proceeding.

An if/else statement takes the following form:

```
if (condition)
    statement1;
else
    statement2;
```

If condition evaluates to true, then statement1 is executed, otherwise statement2 is executed.

The ?: operator takes the following form:

```
(condition) ? expression1 : expression2;
```

If condition evaluates to true, then expression1 is executed, otherwise expression2 is executed. Note that expression2 is not optional.

Consider an if/else statement that looks like this:

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

can be rewritten as:

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

In such uses, the conditional operator can help compact code without losing readability.

Parenthesization of the conditional operator

It is common convention to put the conditional part of the operation inside of parentheses, both to make it easier to read, and also to make sure the precedence is correct. The other operands evaluate as if they were parenthesized, so explicit parenthesization is not required for those.

Note that the ?: operator has a very low precedence. If doing anything other than assigning the result to a variable, the whole ?: operator also needs to be wrapped in parentheses.

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

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

Or we could use the conditional operator to do this:

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

Let's examine what happens if we don't parenthesize the whole conditional operator in the above case.

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

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

would evaluate as:

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

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

Best practice

Always parenthesize the conditional part of the conditional operator, and consider parenthesizing the whole thing as well.

The conditional operator evaluates as an expression

Because the conditional operator operands are expressions rather than statements, the conditional operator can be used in some places where if/else can not.

For example, when initializing a constant variable:

```
#include <iostream>

int main()
{
    constexpr bool inBigClassroom { false };
    constexpr int classSize { inBigClassroom ? 30 : 20 };
    std::cout << "The class size is: " << classSize <<
'\n';

    return 0;
}
```

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

```
#include <iostream>

int main()
{
    constexpr bool inBigClassroom { false };

    if (inBigClassroom)
        constexpr int classSize { 30 };
    else
        constexpr int classSize { 20 };

    std::cout << "The class size is: " << classSize <<
'\n';

    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;

    return 20;
}

int main()
{
    const int classSize { getClassSize(false) };
    std::cout << "The class size is: " << classSize <<
'\n';

    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, either the type of both expressions in a conditional statement must match, or the both expressions must be convertible to a common type.

> **For advanced readers**
>
> The conversion rules used when the types don't match are rather complicated. You can find them here (https://en.cppreference.com/w/cpp/language/operator_other).

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

```
#include <iostream>

int main()
{
    constexpr 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 C-style string literal. The compiler is unable to determine a common type for expressions of these types. In such cases, you'll have to use an if/else.

So when should you use the conditional operator?

The conditional operator gives us a convenient way to compact some if/else statements. It's most useful when we need a conditional initializer (or assignment) for a variable, or to pass a conditional value to a function.

It should not be used for complex if/else statements, as it quickly becomes both unreadable and error prone.

> **Best practice**
>
> Only use the conditional operator for simple conditionals where you use the result and where it enhances readability.

Next lesson
5.6 Relational operators and floating point comparisons

Back to table of contents

Previous lesson
5.4 Increment/decrement operators, and side effects

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment!

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies:

POST COMMENT

199 COMMENTS

Newest