5.5 — Comma and conditional operators

BY ALEX ON JUNE 14TH, 2007 | LAST MODIFIED BY NASCARDRIVER ON FEBRUARY 9TH, 2020

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); // 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 a
```

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;

    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 **5.7 -- 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
    int z(3), w(5); // Comma used to separate multiple variables being defined on the same line
}
```

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** 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 parenthesis, 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 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 use the conditional operator to do this:

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

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;
```

would evaluate as:

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

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 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()
```

```
4  {
5      int x = 5;
6      std::cout << (x != 5 ? x : "x is 5"); // won't compile
7
8      return 0;
9  }
```

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.

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.

 **5.6 -- Relational operators and floating point comparisons**

 **Index**

 **5.4 -- Increment/decrement operators, and side effects**

C++ TUTORIAL | PRINT THIS POST

99 comments to 5.5 — Comma and conditional operators

« Older Comments 1 2

Michael
February 8, 2020 at 3:56 pm · Reply

I'm confused as to why you are claiming that the spaceship operator is a ternary operator. cppreference seems to show usage of this operator as follows: `x <=> y` (x and y are the only 2 operators).

Here's the link: https://en.cppreference.com/w/cpp/language/operator_comparison#Three-way_comparison

It returns something >0, <0, or ==0. So the "three-way" seems to refer to the output (i.e. it's "one more" than two-way Boolean output).

So even in C++20, there's still only one ternary operator. Is this right?

nascardriver
February 9, 2020 at 7:04 am · Reply

The spaceship operator indeed has only 2 operands. Thanks for pointing out the mistake, lesson amended!

Tom Bauer
February 1, 2020 at 2:09 pm · Reply

Would this be a reliable workaround to the last problem?

```
1 int main()
2 {
3     int x{5};
4     ((x != 5) ? (std::cout << x) : (std::cout << "x is 5"));
5 }
```

nascardriver
February 2, 2020 at 1:13 am · Reply

Yes, but that's not a recommended way of using the conditional operator. Only use the conditional operator if you use it's result. Use an if-statement otherwise.

Jake
December 9, 2019 at 6:10 pm · Reply

In the last example:
std::cout << (x != 5 ? x : "x is 5"); // won't compile
You go on to explain that:
One of the expressions is an integer, and the other is a string literal.

How is the 5 a string literal? It looks like a perfectly legitimate integer literal to my eyes.

What am I missing?

nascardriver

December 11, 2019 at 4:36 am · Reply

```
(x != 5 ? x : "x is 5")
 ^^^^^^ bool, irrelevant

(x != 5 ? x : "x is 5")
          ^ int

(x != 5 ? x : "x is 5")
              ^^^^^^^^ string literal
```

The two possible results must be the same or one must be convertible to the other.

Matt

August 4, 2019 at 5:07 am · Reply

Wow, more info regarding the language that my Intro course 'conveniently' left out. I am beginning to see just how much in terms of concepts, tools, and best practices were completely omitted from the assigned C++ textbook. Unlearning the bad habits the book taught is going to take some work.

(If your curious, the textbook I was assigned was "C++ Programming: From Problem Analysis to Program Design" by D.S. Malik, and the very first thing it does it teaches its students use namespace std instead scope reducing with std::, and the professor never offered any feedback for improving code in any way.)

nascardriver

August 4, 2019 at 8:18 am · Reply

Yep, a lot of teaching resources teach `using namespace std;`, because it's convenient and makes to code smaller. But even if they say that it's bad, when they use it, their students will too.

Ikenna

December 5, 2019 at 1:35 pm · Reply

I have a question if you don't mind me asking. Has a lot changed in the C++ language architecture to start to worry about not reading books on C++ that has been dated since 2010 ?

nascardriver

December 6, 2019 at 2:17 am · Reply

Yes, a lot has changed since 2010. You can read the book if it's about a specific topic, eg. algorithms, networking, etc., but I can't recommend reading it to learn the language.
If you're planning on using an old standard (For embedded system with an old compiler for example), you can also read the book. However, it's easier to learn the current language version and stop using new features than it is to learn an old language version and read up on everything you missed.

Benur21
July 29, 2019 at 1:39 pm · Reply

I think it's worth noting that the else part on these ?: statements is not optional like in the if's, because it always must return something. Sometimes we could use empty string "" if we want it to return nothing. For example, in the following code I found at lesson 5.5:

```
#include <iostream>

// Iterate through every number between 1 and 50
int main()
{
    int count = 1;
    while (count <= 50)
    {
        // print the number (pad numbers under 10 with a leading 0 for formatting purposes)
        if (count < 10)
            std::cout << "0" << count << " ";
        else
            std::cout << count << " ";

        // if the loop variable is divisible by 10, print a newline
        if (count % 10 == 0)                  // <----------- LOOK AT THIS ONE
            std::cout << "\n";                // <----------- LOOK AT THIS ONE

        // increment the loop counter
        ++count;
    }

    return 0;
}
```

I was thinking if I could do like this:

```
#include <iostream>

// Iterate through every number between 1 and 50
int main()
{
    int count = 1;
    while (count <= 50)
    {
        // print the number (pad numbers under 10 with a leading 0 for formatting purposes)
        if (count < 10)
            std::cout << "0" << count << " ";
        else
            std::cout << count << " ";

        // if the loop variable is divisible by 10, print a newline
        std::cout << (count % 10 == 0) ? "\n";          // <----------- LOOK AT THIS ONE

        // increment the loop counter
        ++count;
    }

    return 0;
}
```

but obviously it needs an else return.

Alex
August 3, 2019 at 3:22 pm · Reply

Good catch. I'm in the process of rewriting this lesson and will include a note about this. Thanks!

Randle
June 10, 2019 at 8:37 am · Reply

I remember learning the conditional operator in uni, but was never shown a proper example where it was necessary, so I ignored it. Thank you for providing a clear example of its usage!

Brandon
March 14, 2019 at 9:43 am · Reply

How could I use the conditional operator in place of an if/else with multiple statements? I'm practicing by converting if/else statements to use the conditional operator but what about those with multiple else if's like a switch statement? Is that possible? or would it be bad practice since the point is to reduce the clutter of code? Thanks.

EDIT: I just noticed " It's worth noting that the conditional operator evaluates as an expression, whereas if/else evaluates as a set of statements."

But still, can I only evaluate a single expression at once right? Sorry if this sounds weird.

nascardriver
March 15, 2019 at 5:40 am · Reply

Hi Brandon!

The conditional operator should only be used when you need the result of the expression.

```
1  int i{};
2
3  if (something)
4  {
5    i = 3;
6  }
7  else
8  {
9    i = 9;
10 }
11
12 // should be replaced with
13 int i{ something ? 3 : 9 };
14
15 // But
16
17 int i{};
18
19 if (something)
20 {
21   i += 9;
22 }
23 else
24 {
25   something = true;
26 }
```

```
27
28  // should NOT be replaced with
29  something ? (i += 9) : (something = true);
```

You could use the comma operator to get what you want

```
1  something ? ((i += 9), (something = true), (std::cout << "wow\n"), false) : (something
```

But don't do it, it's ugly, use if-statements.

Jules
January 26, 2019 at 1:25 am · Reply

So i wrote a simple assembly program to find the square of a number (2 in this case, haven't given the functionality of user input):

```
#include "pch.h"
#include <iostream>
int power(int);
int main()
{
   std::cout << "Hello World!\n";
   std::cout<<power(2);
}

int power(int x)
{
   _asm
   {
      mov eax,x
      imul eax,x
      mov x,eax

   }
   return x;

}
```

my question is :
in the instruction mov x,eax does the compiler already know that the contents of eax should be moved into the MEMORY LOCATION of "x"
because the contents of the eax register cannot be moved into a variable.

consequently mov [x],eax works as well.
just wanted to know if the c++ compiler is optimized in handling asm in a way that traditional assemblers like NASM or FASM aren't.

Jeremy
January 17, 2019 at 2:36 am · Reply

I can understand the logic of variables inside of functions being destroyed after the function call is over, but is there a specific reason why if/else statements share the same scope?

Alex
January 21, 2019 at 11:25 am · Reply

The C++ specification says that if the if-statement or else-statement are not written as compound statements, they should be treated as if they were. So:

```
if (x)
    int y;
```

Is treated as if it were written:

```
if (x)
{
    int y;
}
```

And although we haven't covered blocks/compound statements yet (that happens in chapter 4), they behave similarly to functions in that local variables defined inside a block die at the end of the block.

aleksandr.a
December 15, 2018 at 9:22 am · Reply

So you can use the conditional operator?
Thank.

```
#include <iostream>

using namespace std;

int main(){

    //Example (with conditional operator): Output even and odd numbers.
    cout << "Even\tOdd\n";
    for (int i{}, j{}; i < 51; i += ((!(i % 2)) ? 2 : 1), j += ((!(j % 2)) ? 1 : 2)) {
    // or this :
    //for (int i{}, j{}; i < 51; i += ((i%2) ? 1 : 2), j += ((j%2) ? 2 : 1)) {
       if(i == 0 && j == 0){
           continue;
       }
    cout << setw(2) << i << " .... " << j << endl;
    }

    cin.clear();
    cin.ignore();
    cin.get();

    return 0;
}
```

Rai
November 15, 2018 at 12:04 pm · Reply

I tried using the conditional operator with one of the older lesson project.

```
int getInteger()
{
    std::cout << "Enter a single digit integer: " << std::endl;
    int input;
    std::cin >> input;
    if (input > 0)
    {
        if (input < 10)
            return input;
```

```
    }
    return -1;
}
```

How would you use conditional operators for the if statement here? I tried doing it with this :

```
int checkInput{ ((input > 0) ? ((input < 10) ? input : 0) : -1) };
```

But I was getting an error saying "error C4716: 'getInteger': must return a value"

nascardriver
November 15, 2018 at 12:12 pm · Reply

Without the full @getInterger function I can't tell you what your problem is.
Here's how you could do it

```
int getInteger(void)
{
  std::cout << "Enter a single digit integer: " << std::endl;
  int input{ 0 };
  std::cin >> input;

  return (((input > 0) && (input < 10)) ? input : -1);
}
```

leonidus007
June 17, 2019 at 7:42 am · Reply

Maybe this can help..

```
#include <iostream>
int getInteger()
{
   std::cout << "Enter a single digit integer: " << std::endl;
   int input;
   std::cin >> input;
   return input > 0 ? (input < 10 ? input : -1) : -1;
}
int main()
{
   int y{ getInteger() };
   std::cout << y;
   return 0;
}
```

« Older Comments 1 2