5.6 — The conditional operator

ALEX JANUARY 18, 2024

Operator	Symbol	Form	Meaning
Conditional	?:	c ? x : y	If conditional `c` is `true` then evaluate `x`, otherwise evaluate `y`

The **conditional operator** (`?:`) (also sometimes called the **arithmetic if** operator) is a ternary operator (an operator that 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.

Related content

We cover if-else statements in lesson 4.10 -- Introduction to if statements (https://www.learncpp.com/cpp-tutorial/introduction-to-if-statements/).

To recap, 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 `else` and `statement2` are optional.

The `?:` operator takes the following form:

```
condition ? expression1 : expression2;
```

If `condition` evaluates to `true`, then `expression1` is evaluated, otherwise `expression2` is evaluated. The `:` and `expression2` are not optional.

Consider an if-else statement that looks like this:

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

This can be rewritten as:

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

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

The conditional operator evaluates as an expression

Because the operands of the conditional operator are expressions rather than statements, the conditional operator can be used in places where an expression is required.

For example, when initializing a 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 direct if-else replacement 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-statement or else-statement die at the end of the if-statement 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 << '\n';

    return 0;
}
```

This one works because `getClassSize(false)` is an expression, and the if-else logic is inside a function (where we can use statements). But this is a lot of extra code when we could just use the conditional operator instead.

Parenthesizing the conditional operator

Because C++ prioritizes the evaluation of most operators above the evaluation of the conditional operator, it's quite easy to write expressions using the conditional operator that don't evaluate as expected.

> **Related content**
>
> We cover the way that C++ prioritizes the evaluation of operators in future lesson 6.1 -- Operator precedence and associativity (https://www.learncpp.com/cpp-tutorial/operator-precedence-and-associativity/).

For example:

```
#include <iostream>

int main()
{
    int x { 2 };
    int y { 1 };
    int z { 10 - x > y ? x : y };
    std::cout << z;

    return 0;
}
```

You might expect this to evaluate as `10 - (x > y ? x : y)` (which would evaluate to `8`) but it actually evaluates as `(10 - x) > y ? x : y` (which evaluates to `2`).

For this reason, the conditional operator should be parenthesized as follows:

- Parenthesize the entire conditional operator when used in a compound expression (an expression with other operators).
- For readability, consider parenthesizing the condition if it contains any operators (other than the function call operator).

The operands of the conditional operator do not need to be parenthesized.

Let's take a look at some statements containing the conditional operator and how they should be parenthesized:

```
    return isStunned ? 0 : movesLeft;           // not used in compound expression, condition contains no operators
    int z { (x > y) ? x : y };                  // not used in compound expression, condition contains operators
    std::cout << (isAfternoon() ? "PM" : "AM"); // used in compound expression, condition contains no operators (function call
    operator excluded)
    std::cout << ((x > y) ? x : y);             // used in compound expression, condition contains operators
```

Best practice

Parenthesize the entire conditional operator when used in a compound expression.

For readability, consider parenthesizing the condition if it contains any operators (other than the function call operator).

The type of the expressions must match or be convertible

To comply with C++'s type checking rules, one of the following must be true:

- The type of the second and third operand must match.
- The compiler must be able to find a way to convert one or both of the second and third operands to matching types. The conversion rules the compiler uses are fairly complex and may yield surprising results in some cases.

For example:

```
    #include <iostream>

    int main()
    {
        std::cout << (true ? 1 : 2) << '\n';    // okay: both operands have matching type int

        std::cout << (false ? 1 : 2.2) << '\n'; // okay: int value 1 converted to double

        std::cout << (true ? -1 : 2u) << '\n';  // surprising result: -1 converted to unsigned int, result out of range

        return 0;
    }
```

Assuming 4 byte integers, the above prints:

```
1
2.2
4294967295
```

In general, it's okay to mix operands with fundamental types (excluding mixing signed and unsigned values). If either operand is not a fundamental type, it's generally best to explicitly convert one or both operands to a matching type yourself so you know exactly what you'll get.

Related content

The surprising case above related to mixing signed and unsigned values is due to the arithmetic conversion rules, which we cover in lesson 10.5 -- Arithmetic conversions (https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/).

If the compiler can't find a way to convert the second and third operands to a matching type, a compile error will result:

```
#include <iostream>

int main()
{
    constexpr int x{ 5 };
    std::cout << ((x != 5) ? x : "x is 5"); // compile error: compiler can't find common type for constexpr int and C-style
string literal

    return 0;
}
```

In the above example, one of the expressions is an integer, and the other is a C-style string literal. The compiler will not be able to find a matching type on its own, so a compile error will result.

In such cases, you can either do an explicit conversion, or use an if-else statement:

```
#include <iostream>
#include <string>

int main()
{
    int x{ 5 }; // intentionally non-constexpr for this example

    // We can explicitly convert the types to match
    std::cout << ((x != 5) ? std::to_string(x) : std::string{"x is 5"}) << '\n';

    // Or use an if-else statement
    if (x != 5)
        std::cout << x << '\n';
    else
        std::cout << "x is 5" << '\n';

    return 0;
}
```

For advanced readers

If `x` is constexpr, then the condition `x != 5` is a constant expression. In such cases, using `if constexpr` should be preferred over `if`, and your compiler may generate a warning indicating so (which will be promoted to an error if you are treating warnings as errors).

Since we haven't covered `if constexpr` yet (we do so in lesson 8.4 -- Constexpr if statements (https://www.learncpp.com/cpp-tutorial/constexpr-if-statements/)), `x` is non-constexpr in this example to avoid the potential compiler warning.

So when should you use the conditional operator?

The conditional operator is most useful when doing one of the following:

- Initializing an object with one of two values.
- Assigning one of two values to an object.
- Passing one of two values to a function.
- Returning one of two values from a function.
- Printing one of two values.

Complicated expressions should generally avoid use of the conditional operator, as they tend to be error prone and hard to read.

Best practice

Prefer to avoid the conditional operator in complicated expressions.

Next lesson
5.7 Inline functions and variables

Back to table of contents

Previous lesson
5.5 Constexpr variables

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

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

27 COMMENTS

Newest