1.9 — Introduction to expressions

BY ALEX ON JUNE 6TH, 2007 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 25TH, 2020

Expressions

Consider the following series of statements:

```
int x{ 2 };             // initialize variable x with value 2
int y{ 2 + 3 };         // initialize variable y with value 5
int z{ (2 * 3) + 4 };   // initialize variable z with value 10
int w{ y };             // initialize variable w with value 5 (the current value of variable y)
```

Each of these statements defines a new variable and initializes it with a value. Note that the initializers shown above make use of a variety of different constructs: literals, variables, and operators. Somehow, C++ is converting each of these literals, variables, and operators into a single value that can be used as the initialization value for the variable.

What do all of these have in common? They make use of an expression.

An **expression** is a combination of literals, variables, operators, and explicit function calls (not shown above) that produce a single output value. When an expression is executed, each of the terms in the expression is evaluated until a single value remains (this process is called **evaluation**). That single value is the **result** of the expression.

Here are some examples of different kinds of expressions, with comments indicating how they evaluate:

```
2               // 2 is a literal that evaluates to value 2
"Hello world!"  // "Hello world!" is a literal that evaluates to text "Hello world!"
x               // x is a variable that evaluates to the value of x
2 + 3           // 2 + 3 uses operator + to evaluate to value 5
x = 2 + 3       // 2 + 3 evaluates to value 5, which is then assigned to variable x
std::cout << x  // x evaluates to the value of x, which is then printed to the console
```

As you can see, literals evaluate to their own values. Variables evaluate to the value of the variable. We haven't covered function calls yet, but in the context of an expression, function calls evaluate whatever value the function returns. And operators let us combine multiple values together to produce a new value.

Note that expressions do not end in a semicolon, and cannot be compiled by themselves. For example, if you were to try compiling the expression *x = 5*, your compiler would complain (probably about a missing semicolon). Rather, expressions are always evaluated as part of statements.

For example, take this statement:

```
int x{ 2 + 3 }; // 2 + 3 is an expression that has no semicolon -- the semicolon is at the end
```

If you were to break this statement down into its syntax, it would look like this:

```
type identifier { expression };
```

Type could be any valid type (we chose *int*). *Identifier* could be any valid name (we chose *x*). And *expression* could be any valid expression (we chose *2 + 3*, which uses 2 literals and an operator).

Key insight

Wherever you can use a single value in C++, you can use an expression instead, and the compiler will resolve the expression down to a single value.

Expression statements

Certain expressions (like *x = 5*) are useful by themselves. However, we mentioned above that expressions must be part of a statement, so how can we use these expressions by themselves?

Fortunately, we can convert any expression into an equivalent statement (called an expression statement). An **expression statement** is a statement that consists of an expression followed by a semicolon. When the statement is executed, the expression will be evaluated (and the result of the expression will be discarded).

Thus, we can take any expression (such as *x = 5*), and turn it into an expression statement (such as *x = 5;*) that will compile.

Note that we can make expression statements that compile but are meaningless/useless (e.g. *2 * 3;*). This expression evaluates to 6, and then the value 6 is discarded.

Rule

Values calculated in an expression are discarded at the end of the expression.

Quiz time

Question #1

What is the difference between a statement and an expression?

Show Solution

Question #2

Indicate whether each of the following lines are *statements that do not contain expressions*, *statements that contain expressions*, or are *expression statements*.

a)

```
1 int x;
```

Show Solution

b)

```
1 int x = 5;
```

Show Solution

c)

```
1 x = 5;
```

Show Solution

d) Extra credit:

```
std::cout << x; // Hint: operator<< is a binary operator.
```

Show Solution

Question #3

Determine what values the following program outputs. Do not compile this program. Just work through it line by line in your head.

```
#include <iostream>

int main()
{
    std::cout << 2 + 3 << '\n';

    int x{ 6 };
    int y{ x - 2 };
    std::cout << y << '\n';

    int z{ 0 };
    z = x;
    std::cout << z - x << '\n';

    return 0;
}
```

Show Solution

 **1.10 -- Developing your first program**

 **Index**

 **1.8 -- Introduction to literals and operators**

C++ TUTORIAL | PRINT THIS POST

120 comments to 1.9 — Introduction to expressions

« Older Comments 1 2

Vitaliy Sh.
January 24, 2020 at 10:02 pm · Reply

Hi Sires!

In question #3: no space in #include. Doesn't prevent compilation, though.

Misc:
"type identifier { expression };
... (we chose 2 + 3, which uses 2 literals and an operator) ..."
** ``literals (type-int)``
** It's to reference a previous lesson 1.8: "Literals and variables both have a value (and a type)."

"Expressions are used when we want the program to calculate a value."
** I'd read the https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/comment-page-1/#comment-251563
** Question 2d: std::cout and x produced some "stream" value. Isn't this a "statement contains an expression" (because it's also prints)?

Jim
December 12, 2019 at 10:42 am · Reply

Alex
I believe the note in this lesson is incorrect. Since in the statement: int z{ (2 * 3) + 4 };. The expression is actually:
{(2 * 3) + 4 }. And since you never really defined what a expression actually is.

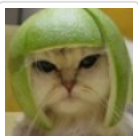
nascardriver
December 13, 2019 at 2:15 am · Reply

The curly braces are a part of the braced-init-list, not of the expression. Here's a breakdown of the grammar:

```
simple-declaration(int z{ (2 * 3) + 4 })
decl-specifier-seq(int) init-declarator(z{ (2 * 3) + 4 })
decl-specifier-seq(int) declarator-id(z) braced-init-list({ (2 * 3) + 4 })
decl-specifier-seq(int) declarator-id(z) { additive-expression((2 * 3) + 4) }
```

trisha
November 15, 2019 at 12:30 am · Reply

in your quiz, The question #2 □x = 5□why did you said that the ‘5’is an expression.
isn't it a value?



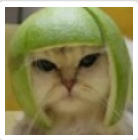
Alex
November 15, 2019 at 4:16 pm · Reply

It's actually both. In the lesson, it says, "An [def]expression[/def] is a combination of literals, variables, operators, and explicit function calls (not shown above) that produce a single output value"

In this case, the expression is just a single literal, that produces a single output value (the value of the literal).

trisha
November 14, 2019 at 5:22 am · Reply

what is the difference between expression and value?



Alex
November 15, 2019 at 2:59 pm · Reply

A value is a a discrete piece of data that can be stored in a variable and manipulated by a program. Example values: 4, 'a', "Hello".

An expression is a combination of values, variables, operators, and function calls that is evaluated to produce a value. Example expressions: 2 + 3, x, x + 4

Michael J. Kelly
October 31, 2019 at 1:29 pm · Reply

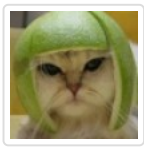
Hello Alex,

First, let me join the throng of those who appreciate the generosity and excellence of this tutorial. What a find!

I was hoping you could clarify a point. In your text about expressions, you list the variable 'x' among the types of expressions, noting that it "evaluates to the value of x" in your description. If I accept that (I do), then I would have answered question (a) as being a statement that contains an expression (one that evaluates to the value of x).

Again, many thanks for your quality work on this tutorial. I am using it as a primer to get ready to work with my son and some classmates in Vex Robotics. I am a programmer, but not in C++ (well, not since a couple of courses way back in 1998), so this is exactly what I needed to break ground.

Best regards,
Mike Kelly



Alex
November 1, 2019 at 10:46 pm · Reply

In the context of 2c and 2d, x is evaluated to produce a value, so it is part of an expression. In the context of 2a, x is _not_ being evaluated (there's no evaluation at all here) -- rather, x is being defined. Definition happens as part of a statement, not an expression.

Michael Kelly
November 4, 2019 at 4:05 pm · Reply

That makes perfect sense. On my first read I overlooked the fact that the example in the problem was a definition statement. Thanks Alex.

Fernando Rosendo
August 18, 2019 at 11:55 am · Reply

Hey, I might be wrong, but, in regards to "Rule - Values calculated in an expression are discarded at the end of the expression.", doesn't that only apply to some cases?

For example:

```
1 int x = 5 + 3;
```

That is a statement with an expression, right? And it means that 5 + 3 will be evaluated to 8, which will be assigned to x. What is the general rule then, if I'm correct? When is the "5+3" evaluation discarded?

My guess is that the evaluation result of an expression is discarded IF there are no operators nearby (then it could be defined as Exp -> Exp OP Exp)?

Alex
August 18, 2019 at 6:42 pm · Reply

First 5+3 evaluates to the value 8. Then 8 is used to initialize variable x. Then 8 is discarded.

Fernando
August 19, 2019 at 11:23 am · Reply

Thanks, Alex!

Bella
July 9, 2019 at 8:40 am · Reply

What do you mean by:

"Note that we can make expression statements that compile but are meaningless/useless (e.g. 2 * 3;). This expression evaluates to 6, and then the value 6 is discarded.
Rule - Values calculated in an expression are discarded at the end of the expression."

Does this basically mean its not good to use expression as the value in the end will be discarded anyway?

Alex
July 9, 2019 at 11:11 am · Reply

No. The point is that expression statements can be useful or useless.

```
2 * 3 ; // useless because the calculated value (6) is discarded before doing anything
x = 2 * 3; // useful, because the calculated value (6) is used.
```

In the latter case, the value of 6 is still discarded at the end of the expression, but that's fine, because we've already assigned it to x for use later.

Bella
July 10, 2019 at 6:40 am · Reply

I get it now, thanks Alex!

Mabel
July 6, 2019 at 8:06 am · Reply

"Expressions don't end in a semicolon & cannot be compiled by themselves.
E.g. If you were to compile the expression x = 5, your compiler would complain about a missing semicolon.
Rather, expressions are always evaluated as part of statements."

Does this mean all Expressions have to be evaluated in the same structure as a statement?

e.g "Hello world!" // Expression
std::cout <<"hello world!"; // Statement

e.g
2 + 3 // Expression
int x{2 + 3}; // Statement

Alex
July 8, 2019 at 10:32 am · Reply

I'm not sure what you mean by "in the same structure" -- but generally, statements can contain one or more expressions, which are evaluated when the statement evaluates.

Mabel
July 9, 2019 at 12:40 am · Reply

I guess I'm just a bit confused on the whole meaning of expression and statements. Reading through your examples (listed below), I noticed that each of the expressions didn't have a
semicolon nor did they have a datatype 'int' for example.

"Examples of different kinds of expressions, with comments indicating how they evaluate:"
1 2 // 2 is a literal that evaluates to value 2
2 "Hello world!" // "Hello world!" is a literal that evaluates to text "Hello world!"
3 x // x is a variable that evaluates to the value of x
4 2 + 3 // 2 + 3 uses operator + to evaluate to value 5
5 x = 2 + 3 // 2 + 3 evaluates to value 5, which is then assigned to variable x
6 std::cout << x // x evaluates to the value of x, which is then printed to the console

... We also learnt about statements in this lesson.
The examples you listed below all included a datatype and a semicolon.

```
int x{ 2 };              // initialize variable x with value 2
int y{ 2 + 3 };          // initialize variable y with value 5
int z{ (2 * 3) + 4 };  // initialize variable z with value 10
int w{ y };              // initialize variable w with value 5 (the current value of variable y)
```

Now, when we write an expression on Visual Studio, it comes up as an error - mainly due to the exclusion of a semicolon. When I wrote "in the same structure", I was asking if we have to change the structure of an expression e.g (2 + 3) to the structure of a statement such as e.g. ---- int x{2 + 3}; ---- so no errors would pop up on Visual Studio - as statements use semicolons therefore no errors would pop up.

I hope that makes sense.

Alex
July 9, 2019 at 11:03 am · Reply

You can take any expression, add a semicolon, and it will become an expression statement that will compile. e.g.

```
2 + 3; // this is an expression statement, albeit a useless one
```

So no, you don't have to change it, but expressions and statements have different intents -- expressions are evaluated to produce a value, and statements perform some task.

Mabel
July 10, 2019 at 6:02 am · Reply

Thank you, thank you thank you Alex!! You are actually the best.

Destiny
July 3, 2019 at 7:31 am · Reply

Are expression and process of evaluation technically the same thing?
And could you specify what you mean by "Term"?
When an expression is carried out, each "term" in the expression is evaluated until a single value remains (this process is evaluation).

nascardriver
July 4, 2019 at 5:59 am · Reply

An expression is a syntactical construct, ie. something you write down. Evaluation is the process of executing (Getting the value of) an expression.

Destiny
July 4, 2019 at 6:45 am · Reply

Nascar, you said previously that:
"An expression is a syntactical construct, ie. something you write down.
Evaluation is the process of executing (Getting the value of) an expression."

Now that I know the difference between an Expression and Evaluation, would you mind helping me out to understand the difference between expression and statement?
This whole time, I thought an expression (something you write down) was a statement.
Isn't a statement something you write down e.g. int y{ 2 + 3 }; //This example is used above for a statement

nascardriver
July 4, 2019 at 7:03 am · Reply

An expression is something you write down, but not everything you write down is an expression.
Every expression is a statement, but not every statement is an expression.
I'd like to refer you to cppreference at this point, they show several examples of expressions and statements:
https://en.cppreference.com/w/cpp/language/expressions
https://en.cppreference.com/w/cpp/language/statements

Destiny
July 4, 2019 at 8:11 am · Reply

Wait, either I've got it wrong the whole time, or my words have misled you...
EXPRESSION
You wrote previously "An expression is a syntactical construct, ie. something you write down"
Yet above, the answer in Q1. was "Expressions are used when we want the program to calculate a value.

nascardriver
July 4, 2019 at 8:13 am · Reply

I don't see the problem, mind elaborating?

Amanda
July 3, 2019 at 5:58 am · Reply

We haven't talked much about function calls... just wondering, is an example of function call in the below.

```
1  2             // 2 is a literal that evaluates to value 2
2  "Hello world!"  // "Hello world!" is a literal that evaluates to text "Hello world!"
3  x             // x is a variable that evaluates to the value of x
4  2 + 3         // 2 + 3 uses operator + to evaluate to value 5
5  x = 2 + 3     // 2 + 3 evaluates to value 5, which is then assigned to variable x
6  std::cout << x  // x evaluates to the value of x, which is then printed to the console
```

nascardriver
July 3, 2019 at 6:21 am · Reply

Yes, 6 calls `operator<<` (which is a function) of `std::cout`.
Whether or not 5 is a function call depends on the type of `x`. If `x` is a fundamental type (eg. `int`), this is not a function call.
1-4 are no function calls.

Amanda
July 3, 2019 at 7:45 am · Reply

Thank you heaps!

Amanda
July 6, 2019 at 7:53 am · Reply

Are 'function calls' just another word for functions? (in a expression statement).

And you've stated above that "if x` is a fundamental type (eg. `int`), this is not a function call". Would you be able to give me another example - but this time on how it would be a function call.

ANDDDDD you said before that - 6 calls `operator<<` (which is a function) of `std::cout`.
Does this mean (operator<<) is the function or
(std::cout) is the function?

nascardriver
July 6, 2019 at 8:32 am · Reply

> Are 'function calls' just another word for functions?
No

```
1 void fn(){} // Function
2
3 fn(); // Function call
```

> Would you be able to give me another example - but this time on how it would be a function call.
No, it's too soon. You'll later learn how to overload operators, ie. change what they do. That way you can use `operator+` to concatenate strings for example.

> Does this mean (operator<<) is the function or (std::cout) is the function?
`operator<<` is a function inside of `std::cout`.

alfonso
April 26, 2019 at 11:34 pm · Reply

Would be nice some "deep" look into that "discarded".

"When the statement is executed, the expression will be evaluated (and the result of the expression will be discarded)."

```
1 std::cout << a + b << '\n';
```

Do the result of a + b have a place in RAM, and that place is then available? Or maybe the result resides only in CPU, or ...

Also

```
1 std::cout << 1 + 2 << '\n';
```

I know that the program (.exe) resides in memory so 1 and 2 must be there too. But the result, aka 3 is it in memory for a short period of time?

Sorry I cannot be more precise, my mind is a little fuzzy on this subject :)

nascardriver
April 27, 2019 at 3:50 am · Reply

Both questions are up to the compiler, there is no standard answer. Here's what's commonly done.

1)
Assuming @a and @b are run-time values. If their values aren't on the CPU yet, they will be transferred there. The addition is performed and the result is passed to @std::cout::operator<<. Depending on how large @a, @b and the result of the addition are, and which calling convention your compiler is using, the result may stack on the CPU or is pushed onto the stack (RAM). If the value was pushed onto the stack, it will be popped once @std::cout.operator<< finishes. Note that the stack is not mixed with your other values in RAM. It has it's own reserved space.

2)
1 and 2 are not in memory. The expression was evaluated at compile-time, so a 3 is stored together with the bytes that make up your code, ie. it's not mixed with other values, because code gets its own space in RAM. Depending on the calling convention, 3 is either pushed onto the stack or it stays on the CPU.

Chirayu
February 21, 2019 at 11:51 am · Reply

dude at the end it is written that "we can convert any expression into an equivalent statement" instead it should be "we can convert any expression into an expression statement". Or am I wrong??

Alex
February 22, 2019 at 1:14 pm · Reply

Yep. Clarified the wording in the lesson a bit.

EH
February 5, 2019 at 1:09 pm · Reply

Hello!
First of all, thanks for this great & free tutorial!
I just wanted to add that the answer of question 1 says “A statement are used “. Although this is obviosly just a typo, it is a bit irritating to read- I would replace it with “Statements are used…”

Alex
February 7, 2019 at 6:23 am · Reply

Fixed! Thanks!

NEERAJ SINGH
September 17, 2018 at 1:52 am · Reply

WHAT IS OPERATOR ^

nascardriver
September 17, 2018 at 2:01 am · Reply

Bitwise XOR
https://www.learncpp.com/cpp-tutorial/38-bitwise-operators/

Rostyslav
July 1, 2018 at 11:08 pm · Reply

Good day. Finally I found a good material to start learning C++ for beginners. Before I tried Stroustrup B. - Programming Principles and Practice Using C++, 2nd Edition - 2014 for 1 month. The explanations on this site are quite easy to understand for a newbie, than Stroustrup's. Especially examples which are given here work well, compare to Stroustrup's ones where sometimes you have to guess what to do extra, to make them work.
Thank you very much for your work!

Theodor
May 26, 2018 at 2:48 pm · Reply

You are the best feel good about your self

Welidien
March 16, 2018 at 6:36 pm · Reply

Hey Alex,

First of all, this website is amazing!

I noticed that there is an operator for addition, subtraction, multiplication, division, and an operator for finding the remainder. However, I did not see an operator for finding the exponent. For example, I could not find a specific operator that does something like this:

2^3 = 8

I did a quick google search and was unable to find such an operator for C++. Referring to this website: https://www.geeksforgeeks.org/operators-c-c/ the '^' means "Bitwise exclusive OR" (I have no idea what that means.) If you have covered this topic in future lessons, please point me to that page.

Thank you. :)

Welidien
March 16, 2018 at 7:39 pm · Reply

Apologies for the question. I found it here: http://www.learncpp.com/cpp-tutorial/32-arithmetic-operators/

Thanks!

Simon Milton
November 11, 2017 at 8:21 pm · Reply

Hello Sir,

This is Simon from Bangalore. Actually i am searching job in Cyber Security and i want to learn C++ for career because i am new to real world experience so that asking this question.

is our website knowledge enough for my field otherwise i want to learn more by books or blogs or any videos ?

Alex
November 13, 2017 at 8:23 pm · Reply

This site will teach you the basics of C++. That will give you the knowledge to learn more about other related subjects. It's definitely not enough to get you a job in cyber security, but it's a good first step along that path.

Arush
July 6, 2017 at 1:43 pm · Reply

Hey Alex,

Just wanted to tell you that lesson 1.6 seems to not exist. Either that or it's giving me a 404 error. Is there an alternate way to get to it?

Thank you.

Alex
July 6, 2017 at 2:07 pm · Reply

We had an issue with the site this morning. It should be fixed now.

Georges Theodosiou
March 22, 2017 at 6:53 am · Reply

My dear c++ Teacher,
Please add std::endl at the end of lines 6 and 7. I mean that

```
1  #include
2
3  int main()
4  {
5      int x = 2; // x is a variable, 2 is a literal
6      std::cout << 3 + 4 << std::endl; // 3 + 4 is an expression, 3 and 4 are literals
7      std::cout << "Hello, world!" << std::endl; // "Hello, world" is a literal too
8
9      return 0;
10 }
```

With regards and friendship.

African
October 29, 2017 at 7:22 am · Reply

I automatically put std::endl; when rewriting the script :)

Tanya
March 10, 2017 at 7:24 am · Reply

```
{   int a,b;
    float x;
std::cin>>a>>b;
```

```
x=a/b;
std::cout<<x;
  return 0;  }
```

I put the value of a = 10 and b=8, and the answer is given as 1. Why?

Alex
March 11, 2017 at 11:50 am · Reply

Dividing two integers does integer division (the answer is actually 1.25, but the fractional component is dropped leaving you with 1).

We cover this in chapter 3.

Georges Theodosiou
March 24, 2017 at 1:42 am · Reply

My dear c++ teacher,
Please let me correct an error of yours. Ten divided by eight yields 1.25. Every calculator insures it. I'm sorry for am maniac with arithmetic accuracy.
With regards and friendship.

Alex
March 24, 2017 at 4:03 pm · Reply

Quite right. Thanks for the correction.

Andrew
November 15, 2016 at 10:45 am · Reply

Alex, you are the man!!

Quick question - in the code snippet in this lesson, would it be prudent to put "return 0;" before "}" just to be consistent with your style in previous lessons?

Just a thought - don't know if it has any merit.

Best,
Andrew

Alex
November 15, 2016 at 3:08 pm · Reply

Yup, added. Thanks for pointing out the inconsistency.

manuel okeke
November 8, 2016 at 10:19 am · Reply

```
1  #include <iostream>
2  using namespace std;
3  int main ()
4  {
```

```
 cout << "hello world";
return 0;
}
```

Georges Theodosiou
October 31, 2016 at 6:03 am · Reply

My dear c++ Teacher,
Please let me comment that at the end of 6th line, "std::endl" should be added, as follows:

```
#include <iostream>

int main()
{
    int x = 2; // x is a variable, 2 is a literal
    std::cout << 3 + 4 << std::endl; // 3 + 4 is an expression, 3 and 4 are literals
    std::cout << "Hello, world!"; // "Hello, world" is a literal too
}
```

With regards and friendship.

Alex
October 31, 2016 at 10:00 am · Reply

To have nice output, I agree we'd want to output std::endl. But the point of this lesson is to talk about literals, so I'm trying to keep the example as simple and focused as possible.

daku
January 22, 2019 at 4:29 am · Reply

i have read all comments. if you are here to learn, then you won't be making such unnecessary comments.

nascardriver
January 23, 2019 at 9:03 am · Reply

Printing a line feed at the end of the program is good practice. Georges' comment is reasonable.

Matt
October 26, 2016 at 2:49 pm · Reply

Possible typo under "Operators":

"Note: One of the most common mistakes the new programmers make is to confuse the assignment operator (=) with the equality operator (==)."

I think maybe you meant to write "that new programmers" instead of "the new programmers".

Alex
October 26, 2016 at 7:54 pm · Reply

I did. Fixed. Thanks!

Alfred O.
September 11, 2016 at 11:43 pm · Reply

Hey Alex, I'm curious about a few things I'm wondering you could tell me about, concerning expressions. The question itself isn't too bad, it's just difficult to word clearly and brief. After some research, I had come to this conclusion:

When an expression evaluates, a temporary object (often called a "temporary") is created to hold the result. This object exists until the end of the full expression, and then it is discarded. If nothing is done with the result of an expression (i.e., the expression has no side effect) it is generally considered a waste.

^ Not entirely sure if this is true. This is how I envision an expression evaluating:

int x = 0, y = 10, z = 15;
x = y + z;

1. x, y, and z "evaluate" but the order of such is not specified. A temporary object is created to hold each of their values.
2. y (10) and z (15) evaluate to 25, another temporary object is created to hold this value.
3. 25 is copied into x.
4. All temporaries are destroyed, in the reverse order they were created in, when the statement (or expression?) ends. More specifically, a "destructor" is called for them, much like the destructor of a class object when it goes out of scope.

It's possible only one temporary is made, or none. I don't actually know what the truth is, so I come to you :-) I had drawn the conclusion that ALL expressions discard their values in the end (the temporaries created) whether or not their result is used, but I could be--and probably am--wrong about that, too. This might at least explain why an l-value reference cannot bind to something like 5 + 5, but an r-value reference can, one of the reasons I was curious about how expressions are carried out by the machine.

Alex
September 12, 2016 at 2:28 pm · Reply

You're on the right track. Expressions do create temporary values that are discarded if nothing is done with them. However, expressions do have specific orders in which they are evaluated -- we cover this in lesson 3.1.

In the above case, y + z are evaluated first, producing temporary rvalue 25. Then operator= is called, and that value of 25 is assigned to variable x. Done.

Alfred O.
September 13, 2016 at 5:06 am · Reply

Ah! I meant the operands themselves may not evaluate in a specified order (well, I mean not specified by the C++ language, so it is left up to the designers of the compiler itself to decide, or maybe the compiler chooses based on what is more efficient).

I read something out of curiosity, which led to one thing, which led to another thing, and now I'm so far off my tracks I can't remember how I got where I'm at, or where I was. I believe it was a wikipedia article on "Sequence Points," something I saw and took note of. According to that, all operands of expressions, and arguments of functions--which I guess could be thought of as arguments of the function call operator ()--aren't guaranteed to evaluate in any specific order. Which is probably why something like:

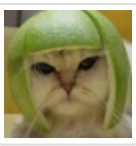
function(i = 5 + x, x = i++, ++y = y - x * i); // Who knows what the heck you'll get???
index[i++] = ++i; // I wouldn't dare to do this anyway

Some places call it "Order of Evaluation" instead of "Sequence Points." And even more confusing, I stumbled across this article while researching it: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0145r1.pdf

Looks like they're trying to change this issue in C++17. If they do, I'll have to get used to that right after learning about it, LOL. I guess it's better they keep improving C++, it's probably worth the headache.

P.S. When the assignment operator = assigns a value, is this value copied into the variable, or is it "bound" like a reference? Or is the temporary value created by evaluation always discarded, regardless of what happens at the end of an expression?

Also, thank you for responding to me. And thank you for this web site! It's made learning the language so much easier. I bought a few of the recommended books (recommended from Stack Oveflow) and they don't really go out of their way to explain much. After going through a few chapters on this site, and THEN reading the book (not all of it, but one day), the book makes perfect sense, so that tells you a whole lot about how good this site is.



Alex
September 13, 2016 at 9:08 am · Reply

Don't worry about sequence points for now. As long as you avoid using a variable with side effects applied more than once in a single expression, you'll be fine. I cover side effects in more detail in chapter 3.

When you use assignment, the value is copied. C++ does support references as well -- we also talk about those in chapter 6.

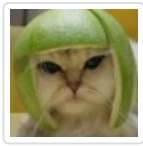
Any temporary values from expressions are discarded at the end of the expression. Whether you copy those values to a variable (via assignment or initialization) is your choice. We talk more about initialization and assignment in chapter 2.

I think it's great that you're going off the rails and doing your own explorations. Just note that a LOT of your questions are probably covered in future lessons, so keep reading!

The reason this site exists is because I found most books to be very difficult to understand if you didn't already know what they were talking about. I try to assume you don't know anything here. Additionally, this site has gone through a lot of refinement based on reader questions, thoughts, and feedback. That's one advantage an online site has over a book!

Alfred O.
September 4, 2016 at 5:16 pm · Reply

Alex, can you explain what a "Primary Expression" is? It's a term I've started to run into since taking a break from learning C++. I come back, and now I'm seeing "Primary Expression," "Postfix Expression," and "l-value" and "r-value" has been appended with "x-value," "gl-value," and "pr-value." It's like I fell through a hole in time-space and woke up in a different dimension. What the heck does all this new stuff mean? Things used to be so simple!



Alex
September 4, 2016 at 7:26 pm · Reply

Yeah, C++11 kind of made a mess of things due to move semantics. From http://en.cppreference.com/w/cpp/language/value_category:

> a glvalue is an expression whose evaluation determines the identity of an object, bit-field, or function
> a prvalue is an expression whose evaluation either computes the value of the operand of an operator (such prvalue has no result object), or initializes an object or a bit-field (such prvalue is said to have a result object). All class and array prvalues have a result object even if it is discarded.
> an xvalue is a glvalue that denotes an object or bit-field whose resources can be reused
> an lvalue is a glvalue that is not an xvalue.
> an rvalue is a prvalue or an xvalue.

There are a bunch of examples of each on that page.

But honestly, I wouldn't sweat about any of them right now except lvalue and rvalue. You can learn about glvalues, prvalues, and xvalues if and when they become useful in some context that's relevant to something you need to know.

Nigel Booth
June 29, 2018 at 4:20 am · Reply

No wonder I'm finding it difficult re-learning C++ using Visual Studio 2017. Last time I touched it was pre-Windows (Borland Turbo C and Turbo C++) a hell of a lot has changed since then. In fact I seem to recall having a listing of the C++ code for the 1st version of Windows somewhere.....

Sam
September 1, 2016 at 8:30 am · Reply

Thanks for the tutorial. Very useful!

Tristan Gybels
June 27, 2016 at 11:34 am · Reply

Hey, Alex!
Try to make a book of this!
I would buy this big time! I'm sure you'll make some big money with it ;)
You only have to cut 'n paste everything.

« Older Comments 1 2