Behaviour of && in C programming language

Asked 11 years, 4 months ago Modified 6 years, 4 months ago Viewed 12k times

20

I am beginner in C programming language, recently I have read about Logical AND `&&` operator.

I also know that, in C programming language all non-zero values are treated as **TRUE**.

```
NON-ZERO && NON-ZERO = 1
NON-ZERO && ZERO = 0
ZERO && NON-ZERO = 0
ZERO && ZERO = 0
```

But when I am dealing with the following program then I am not getting expected answer.

```
int main(){
  int x, y, z;
  x = y = z = -1;
  y = ++x && ++y && ++z;
  printf("x = %d, y = %d, z = %d, x, y, z);
  return 0;
}
```

I am expecting

```
x = 0, y = 0, z = 0
```

but the answer is

```
x = 0, y = 0, z = -1
```

Can anyone please explain, Why I am getting this answer?

Edit: In this question, I have not asked about the precedence of operators.

c operators logical-operators

Share
Improve this question
Follow

edited Aug 18, 2018 at 13:45

asked Aug 14, 2013 at 10:45
Mayank Tiwari
3,024 ● 5 ● 31 ● 53

1 Seems pretty similar to stackoverflow.com/questions/12980757/arithmetic-c-operators – devnull Aug 14, 2013 at 11:02

1 I smell undefined behavior... – fuz Aug 14, 2013 at 11:05

2 @devnull I am not quite sure but short circuit evaluation plus increment operators looks like the ideal recipe to get surprising or undefined behavior. Beware of nasal demons. – fuz Aug 14, 2013 at 11:10

2 @FUZxxl No, shot circuit of `&&` makes sure the calculation sequence of this expression, so no undefined behavior. – Yu Hao Aug 14, 2013 at 11:13

2 @user2320537 An assignment doesn't make this question *unique*. – devnull Aug 14, 2013 at 11:25

5 Answers

Sorted by: Highest score (default)

22

Because of Short-circuit evaluation, when `x` is `0`, `y` and `z` don't really need to be evaluated since `0 && ANYTHING` is `0`.

Once `x` is incremented to `0`, the result is `0`, and that's what `y` gets.

`z` remains unchanged (`-1`).

```
 x  | y  |  z
----+----+-----
 -1 | -1 | -1   //x = y = z = -1;
  0 | -1 | -1   //++x && ... Now the whole expression is evaluated to 0
  0 |  0 | -1   //y = ++x && ++y && ++z;
```

Share Improve this answer Follow

answered Aug 14, 2013 at 10:51
Maroun
95.8k ● 30 ● 193 ● 247

14

I only can think about that `&&` evaluates in **short circuit**: given `A && B`, if `A` evaluates `false` then `B` is not evaluated.

So:

`X` becomes `0`. `&& ++y && ++z` does not evaluates since `X/0/false && ...`

`y=0` as assigned from `y = x/0/false`

`z` remains unmodified since `++z` does not get executed.

Share

Improve this answer

Follow

edited Aug 15, 2013 at 8:57
Grijesh Chauhan
58.2k ● 20 ● 143 ● 213

answered Aug 14, 2013 at 10:50
Alfonso Nishikawa
1,875 ● 1 ● 18 ● 33

4

`&&` operator is evaluated pairwise, therefore I'm guessing C is evaluating

```
((++x && ++y) && ++z)
```

now, `++x` will return zero therefore the first `&&` will fail as will the second one without the need to evaluate `++y` or `++z`.

`y = 0` since that is the result of the expression.

`z` is not touched

Share

Improve this answer

Follow

edited Aug 14, 2013 at 10:58
Maroun
95.8k ● 30 ● 193 ● 247

answered Aug 14, 2013 at 10:53
John Greenall
1,690 ● 13 ● 17

4

What happens is that `++y` and `++z` are never evaluated because the first part already ensures what the new value of `y` will be.

The first part of your statement is `++x && ...` which is equivalent to `0 && ...` and then we already know that `y` will be 0 in the end so the rest of the statement is not executed.

if you did this:

```
int main(){
  int x,y,z,tmp;
  x = y = z = -1;
  tmp = ++x && ++y && ++z;
  printf("x = %d, y = %d, z = %d, tmp = %d", x,y,z, tmp);
  return 0;
}
```

You would get `x = 0, y = -1, z = -1, tmp = 0`

The left evaluation is guaranteed in the C99 standard. You can find it in the section `6.5.13 Logical AND operator`

> Unlike the bitwise binary & operator, the && operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares equal to 0, the second operand is not evaluated.

You can find more information about what is a sequence point on Wikipedia or in the Annex C of the C99 standard

Share

Improve this answer

Follow

edited Aug 14, 2013 at 11:01

answered Aug 14, 2013 at 10:49

koopajah

25.5k ● 9 ● 82 ● 108

4

For completeness (brain dump):

The term behind this sorcery is called short circuiting. Let's go over your code and then a brief blurb about why this happens. Looking at:

```
int main( void ) {
  int x, y, z;
  x = y = z = -1;
  y = ++x && ++y && ++z;

  printf( "x = %d, y = %d, z = %d, x, y, z );

  return 0;
}
```

... we begin to break it down line by line. The first line:

```
int x, y, z;
```

... declares three integers, `x`, `y` and `z`. They are initialized to garbage values on the stack frame because there is no initialization (assignment operator). This line does not really matter, now let's look at the next one:

```
 x = y = z = -1;
```

... we see that we are doing multiple assignments on the same line. Recall that the assignment operator will mutate the identifier to the left of the assignment operator (using the value to the right of the assignment operator) and return the value of `x`. This is known as assignment overloading. But again, this does not really matter -- the only important thing to realize is `x`, `y` and `z` are now all -1. Let's look at the next line:

```
y = ++x && ++y && ++z;
```

... Sorcery Yoda says. Let's add the parenthesis to make it more obvious which step is being evaluated first:

```
y = ( ( ++x ) && ++y && ++z );
```

... now looking at the inner-most parenthesis we see that it's a prefix increment of `x`, meaning we will increment the value of `x` and then return it. We note that `x` is originally -1 and it is now 0 after being incremented. This will resolve as follows:

```
y = ( 0 && ++y && ++z );
```

... now it's important to note that looking at our truth tables:

```
A | B | A && B
--------------
T | T |   T
T | F |   F
F | T |   F
F | F |   F
```

... for the `AND` logical operator we see that both F (AND) T, T (AND) F are F. The compiler realizes this and short circuits when ever it is evaluating a conjunction (AND) where a value is `false` -- a *clever* technique of optimization. It will then resolve to assigning `y` to be 0 (which is false). Recall that in C any non-zero value is `true`, only 0 is `false`. The line will look as follows:

```
y = 0;
```

... now looking at the next line:

```
printf( "x = %d, y = %d, z = %d, x, y, z );
```

... it should be obvious to you now that it will output `x = 0, y = 0, z = -1`.

Share

Improve this answer

Follow

edited Aug 14, 2013 at 11:11

answered Aug 14, 2013 at 11:04

Jacob Pollack

3,751 ● 1 ● 18 ● 39