# 5.2 — Arithmetic operators

BY ALEX ON JUNE 13TH, 2007 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 8TH, 2020

## Unary arithmetic operators

There are two unary arithmetic operators, plus (+), and minus (-). As a reminder, unary operators are operators that only take one operand.

| Operator | Symbol | Form | Operation |
|---|---|---|---|
| Unary plus | + | +x | Value of x |
| Unary minus | - | -x | Negation of x |

The **unary minus** operator returns the operand multiplied by -1. In other words, if x = 5, -x is -5.

The **unary plus** operator returns the value of the operand. In other words, +5 is 5, and +x is x. Generally you won't need to use this operator since it's redundant. It was added largely to provide symmetry with the *unary minus* operator.

For best effect, both of these operators should be placed immediately preceding the operand (e.g. `-x`, not `- x`).

Do not confuse the *unary minus* operator with the *binary subtraction* operator, which uses the same symbol. For example, in the expression `x = 5 - -3;`, the first minus is the *binary subtraction* operator, and the second is the *unary minus* operator.

---

## Binary arithmetic operators

There are 5 binary arithmetic operators. Binary operators are operators that take a left and right operand.

| Operator | Symbol | Form | Operation |
|---|---|---|---|
| Addition | + | x + y | x plus y |
| Subtraction | - | x - y | x minus y |
| Multiplication | * | x * y | x multiplied by y |
| Division | / | x / y | x divided by y |
| Modulus (Remainder) | % | x % y | The remainder of x divided by y |

The addition, subtraction, and multiplication operators work just like they do in real life, with no caveats.

Division and modulus (remainder) need some additional explanation. We'll talk about division below, and modulus in the next lesson.

---

## Integer and floating point division

It is easiest to think of the division operator as having two different "modes".

If either (or both) of the operands are floating point values, the *division operator* performs floating point division. **Floating point division** returns a floating point value, and the fraction is kept. For example, `7.0 / 4 = 1.75`, `7`

`/ 4.0 = 1.75`, and `7.0 / 4.0 = 1.75`. As with all floating point arithmetic operations, rounding errors may occur.

If both of the operands are integers, the *division operator* performs integer division instead. **Integer division** drops any fractions and returns an integer value. For example, `7 / 4 = 1` because the fractional portion of the result is dropped. Similarly, `-7 / 4 = -1` because the fraction is dropped.

**Warning**

Prior to C++11, integer division with a negative operand could round up or down. Thus `-5 / 3` could result in -1 or -2. This was fixed in C++11, which always drops the fraction (rounds towards 0).

## Using static_cast<> to do floating point division with integers

The above raises the question -- if we have two integers, and want to divide them without losing the fraction, how would we do so?

In lesson **4.11 -- Chars**, we showed how we could use the *static_cast*<> operator to convert a char into an integer so it would print as an integer rather than a character.

We can similarly use *static_cast*<> to convert an integer to a floating point number so that we can do *floating point division* instead of *integer division*. Consider the following code:

```
#include <iostream>

int main()
{
    int x{ 7 };
    int y{ 4 };

    std::cout << "int / int = " << x / y << "\n";
    std::cout << "double / int = " << static_cast<double>(x) / y << "\n";
    std::cout << "int / double = " << x / static_cast<double>(y) << "\n";
    std::cout << "double / double = " << static_cast<double>(x) / static_cast<double>(y) << "\

    return 0;
}
```

This produces the result:

```
int / int = 1
double / int = 1.75
int / double = 1.75
double / double = 1.75
```

The above illustrates that if either operand is a floating point number, the result will be floating point division, not integer division.

## Dividing by zero

Trying to divide by 0 (or 0.0) will generally cause your program to crash, as the results are mathematically undefined!

```
#include <iostream>

```

```
int main()
{
    std::cout << "Enter a divisor: ";
    int x{};
    std::cin >> x;

    std::cout << "12 / " << x << " = " << 12 / x << '\n';

    return 0;
}
```

If you run the above program and enter 0, your program will either crash or terminate abnormally. Go ahead and try it, it won't harm your computer.

## Arithmetic assignment operators

| Operator | Symbol | Form | Operation |
|---|---|---|---|
| Assignment | = | x = y | Assign value y to x |
| Addition assignment | += | x += y | Add y to x |
| Subtraction assignment | -= | x -= y | Subtract y from x |
| Multiplication assignment | *= | x *= y | Multiply x by y |
| Division assignment | /= | x /= y | Divide x by y |
| Modulus assignment | %= | x %= y | Put the remainder of x / y in x |

Up to this point, when you've needed to add 4 to a variable, you've likely done the following:

```
x = x + 4; // add 4 to existing value of x
```

This works, but it's a little clunky, and takes two operators to execute (operator+, and operator=).

Because writing statements such as `x = x + 4` is so common, C++ provides five arithmetic assignment operators for convenience. Instead of writing `x = x + 4`, you can write `x += 4`. Instead of `x = x * y`, you can write `x *= y`.

Thus, the above becomes:

```
x += 4; // add 4 to existing value of x
```

 **5.3 -- Modulus and Exponentiation**

 **Index**

 **5.1 -- Operator precedence and associativity**

C++ TUTORIAL | PRINT THIS POST

## 283 comments to 5.2 — Arithmetic operators

« Older Comments 1 2 3 4

Jack
January 7, 2020 at 9:17 am · Reply

Just a grammar point / typo: under "Using static_cast<> to do floating point division with integers" you say the above"begs the question" when I believe you mean it "raises the question." Not a big deal, but I thought I'd mention it since I am finding the course extremely helpful so far! Thanks for putting this all together!

nascardriver
January 8, 2020 at 3:50 am · Reply

This lesson is no longer begging for questions, thanks!

**Juglugs**
October 23, 2019 at 5:10 am · Reply

Did you mean to have the '5' in the sentence?

"Because writing statements such as x = x + 5 is so common, C++ provides 5 arithmetic assignment operators for convenience. Instead of writing x = x + 5, you can write x += 5. Instead of x = x * y, you can write x *= y."

It is confusing because "C++ provides 5 arithmetic assignment operators"

Does C++ provide five operators (if so, you have only shown two) or does C++ provide 5 with some operators?

Alex
October 25, 2019 at 2:08 pm · Reply

Lesson amended. Thanks!

Jon
September 28, 2019 at 1:04 am · Reply

"Trying to divide by 0 (or 0.0) will generally cause your program to crash, as the results are mathematically undefined!"

I hate to be so pedantic, especially since I haven't studied math in 20 years but I always remembered division by zero as infinity. It was a long time ago so I had to check and Wolfram Alpha agrees.
https://www.wolframalpha.com/input/?i=divide%20by%200

I'd recommend that you strike the word mathematically or change it to arithmetically.

Alex
September 30, 2019 at 9:01 am · Reply

Infinity isn't a discrete number, and even if it were, would dividing by 0 produce positive or negative infinity? It's more correct to say it's undefined. See
http://mathforum.org/library/drmath/view/53336.html

**Michael**
October 12, 2019 at 1:05 pm · Reply

I had an issue when I tried to compile the divide by zero example stating this:

"C:\Users\Senith\Desktop\testing\testing\main.cpp|6|error: extended initializer lists only available with -std=c++11 or -std=gnu++11|"

and on ln6 was: int x{}; which i changed to int x; and it compiled with no errors. Any idea why i got that error on codeblocks?

**nascardriver**
October 12, 2019 at 11:33 pm · Reply

You didn't enable a higher standard in code::block's settings.
Brace initialization and many other features aren't available in C++03 (Which is probably what you're using). In your project settings, enable the highest possible C++ standard.

jake
December 8, 2019 at 5:12 pm · Reply

Replying to Jon, posting Sep 28, 2019 at 1:04 AM.

Jon,

As a mathematician wannabe (majored in it) I am conditioned to be that pedantic.  Dividing by zero is an undefined operation.  To emphasize what Alex says, infinity is not a number at all. It is a concept.  It took my first Real Analysis (aka advanced calculus) to rigorously define what is meant by "approaching infinity".

Here's a simple paradox:
If 1 / 0 == infinity then infinity * 0 must == 1.
But 2 / 0 also == infinity, so infinity * 0 must == 2.
Hmm... Then 1 == 2.  EGAD! (Is there a Latin acronym in EGAD, like QED?  :-)

The problem was that I was treating infinity like a number.

(Plugging myself as a Math tutor. :-)

Samira Ferdi
July 9, 2019 at 5:00 pm · Reply

Hi, Alex! If I can, I suggest you to order your operator table in their operator precedence in their related lesson, like your binary arithmetic operators table in this lesson (that is not ordered in their operator precedence although we know that we learn addition and substraction first then multiplication and division).

swagraj
June 28, 2019 at 7:14 am · Reply

is the following func okay for last question

```
bool isEven(int x)
{
   return (!(x % 2)) ;
}
```

**nascardriver**
June 28, 2019 at 12:15 pm · Reply

You can do it and it's always going to work. But `(x % 2) == 0` is easier to read, since you don't have to thing about boolean conversions.

VerticalLimit
June 16, 2019 at 10:48 pm · Reply

Hi Nascardriver & Alex
Quiz 2 - Ive tried to keep it as simple and small as possible. Any suggestions please ?

```
#include<iostream>
bool isEven()
{
    std::cout << "Enter a number & will try to tell you whether its even or odd:";
    int n{0};
    std::cin >> n;
    if (n % 2 == 0 )
        std::cout << n << " its a even number\n";
    else
        std::cout << n << " its a odd number\n";
    return n;
}
int main()
{
    isEven();

    return 0;
}
```

**nascardriver**
June 17, 2019 at 4:35 am · Reply

`isEven` was supposed to have the signature

```
bool isEven(int)
```

You're not using a parameter and your return value doesn't fit the function's name. Try using Alex' `isEven` function and use a conditional operator in `main`. That way you can keep it small.

VerticalLimit
June 17, 2019 at 7:34 am · Reply

Thank you

leonidus007
June 17, 2019 at 4:39 am · Reply

Everything seems good but you might want to crisp short the output statements..

J-Dog
March 25, 2019 at 7:15 pm · Reply

So for Question #1, I got the sequence correct, but don't understand how to apply the % operator apparently. Since 20 % 3 is 6.6666, I should have a result of 6 for the remainder, not 2. I couldn't find the answer to this in the comments below so I must be missing something pretty obvious that no one else got stuck on. Please help.

**nascardriver**
March 26, 2019 at 1:16 am · Reply

20 / 3 = 6 remainder 2
because
6 * 3 = 18
and
18 + 2 = 20

J Dog
March 26, 2019 at 8:14 am · Reply

Thanks. I was looking at it completely wrong. Now it seems simple. Thanks again.

imf
March 1, 2019 at 4:02 am · Reply

bool isOdd(int n) {
    return (n % 2);
}

Arthur
February 28, 2019 at 6:25 pm · Reply

interesting that dividing by zero causes a crash but 0%2 works I also had to test 1%2 thinking it would round down, got stuck thinking on that for a bit. never would have come up with "return x % 2 == 0;"

Dimbo1911
March 26, 2019 at 2:40 am · Reply

You seem to have mixed up few things. Dividing BY zero causes a crash i.e. 7/0 = ????, but dividing THE zero does not i.e. 0/7 = 0 (if you have 0 pieces of a cake and you want to give each of the seven people a piece of a cake, each person recieves 0 pieces of a cake), same with the modules (%), you can module the zero i.e. 0%2 = 0 (0 / 2 = 0 remainder 0, because 0 * 2 = 0), but you cannot module BY zero i.e. 2%0 (undefined behaviour)

Smidge
February 25, 2019 at 11:56 pm · Reply

I have a question regarding the use of count as a variable name in this code. My editor which is code::blocks shows count as green text and I tried to search for what it meant and learnt that it was some kind of input iterator. Is it ok to use it as variable name? It looks confusing to me since my count variable is colored green.

```
#include <iostream>

int main()
{
    // count holds the current number to print
    int count = 1; // start at 1
```

**nascardriver**
February 26, 2019 at 7:23 am · Reply

Hi!
count is the name of a function in the @std namespace. It shouldn't be highlighted unless you include @<algorithm> and are "using namespace std;", but I guess Code::Blocks is using a static list of words to highlight.
You can use it as a variable name.

**mohammad**
February 26, 2019 at 10:37 am · Reply

can you tell me plz why my code for evan or odds leave me with an error

```
 #include <iostream>
int isevan(int x)
{


    if (x % 2 == 0)
        std::cout << x << "is even" << std::endl;
    else
        std::cout << x << "is odd" << std::endl;
}
int main()
{
    std::cout << "enter an integer" << std::endl;
    int number;
    std::cin >> number;
    int b = iseven(number);
    std::cout << b;
```

```
18
19  }
```

**nascardriver**
February 27, 2019 at 3:35 am · Reply

* Line 16: Initialize your variables with brace initializers. You used copy initialization.
* Line 14: Initialize your variables with brace initializers.
* @main, @isevan: Missing return statement.
* Don't use @std::endl unless you have a reason to.

@isevan is declared to return an int, but it doesn't return anything.

**mohammad**
March 5, 2019 at 10:46 am · Reply

thank you so much
that was realy helpful
but can you give me a reference for when to use brace initializers

**nascardriver**
March 6, 2019 at 3:47 am · Reply

Always

Smidge
February 26, 2019 at 6:01 pm · Reply

Great! Thanks! So that's how it works in code::blocks huh? I was really confused as to why it is highlighted even if I didn't include @<algorithm> on my code.

Ryan
February 23, 2019 at 7:39 pm · Reply

Hi, this is my answer for quiz 2.
I was wondering if it is okay or there is a problem? thanks.

```
1   #include <iostream>
2
3   void isEven(int x)
4   {
5       if (x % 2 == 0)
6           std::cout << "The number you entered is an even number. \n";
7       else
8           std::cout << "The number you entered is a odd number. \n";
9   }
10
11  int main()
12  {
13      int x;
14      std::cout << "Enter an integer. \n";
15      std::cin >> x;
```

```
    isEven(x);

    return 0;
}
```

**nascardriver**
February 24, 2019 at 3:18 am · Reply

Hi Ryan!

* Line 13: Initialize your variables with brace initializers.
* "isEven" sounds like the name of a function that returns a value.
* Line 6, 8, 14: Unnecessary space at the end of the line.

J
February 22, 2019 at 12:50 pm · Reply

Hello!

Regarding part 2 of the quiz, by trying to keep every function as simple as possible, I think I may have overthought this. Could you explain the pros and cons to this method?

```
#include <iostream>


void printAnswer(bool isEven)
{
    if (isEven == true)
        std::cout << "Your number is even.";
    else
        std::cout << "Your number is odd.";
}

bool isEven(int number)
{
    return (number % 2 == 0);
}

int getNumber()
{
    std::cout << "Enter an integer: ";

    int number{ };
    std::cin >> number;

    return number;
}

int main()
{
    int number{ getNumber() };

    printAnswer(isEven(number));

    return 0;
}
```

**nascardriver**

February 23, 2019 at 6:26 am · Reply

Hi!

* Line 6: Don't compare booleans to false/true.
* Line 21: Initialize to 0.
* Print a line feed when your program is done.

The structure of your program is fine. If you can split a function into 2 without making your program harder to understand, it's usually a good decision.

J
February 23, 2019 at 10:32 am · Reply

To obey the comment "Print a line feed when your program is done" would it be a better option to omit printAnswer() and add its argument to main() based on the boolean returned from isEven()?

Paulo Filipe
February 7, 2019 at 8:03 am · Reply

Alright, I wrote a program just for fun, to keep writing code and keep teaching the brain how to write c++ that does some mixed division with Integer and Float inputs and outputs either Integer or Float results.

User can choose whether to use static_cast or not to convert integer inputs to float.

I did it because while reading this page, I felt the need to try several division combinations mixing ints with floats, so here it goes.

The program performs the following division combos:

• Int = Int / Int
• Int = Float / Int
• Int = Float / Float
• Float = Int / Int
• Float = Float / Int
• Float = Float / Float

Note that choosing to convert Ints to Floats makes a difference when calculating Float = Int / Int. You'll know why if you read this class paying attention to what you were reading.

For anyone lurking through the comments that wants to test some division combinations, here is the program:

```
1  /*
2   *      Mixed Integer / Float division test
3   */
4
5  #include <iostream>
6
7  using namespace std;
8
9  int intIn()                          // input integer value
10 {
11     cout << "Input int value: \n";
12     int a {0};
13     cin >> a;
14     return a;
15 }
16
```

```
17  float floatIn()                    // input float value
18  {
19      cout << "Input float value: \n";
20      float a {0.0f};
21      cin >> a;
22      return a;
23  }
24
25  // print result with boolean to check if it's integer result
26  void printResult(int intResult, float floatResult, bool isInt)
27  {
28      if (isInt)
29          cout << intResult << ".\n";
30      else
31          cout << floatResult << ".\n";
32  }
33
34  // calculate int = int / int
35  void intDivideIntInt(int useStatic)
36  {
37      cout << "*** Input two integer values to calculate integer division ***\n";
38      int a {intIn()};
39      int b {intIn()};
40      int result {0};         // Integer result
41      if (useStatic == 1)
42          result = static_cast<float>(a) / static_cast<float>(b);
43      else if (useStatic == 2)
44          result = a / b;
45      cout << "Integer division result of integer " << a << " / integer " << b << " is: ";
46      printResult(result, 0, true);
47  }
48
49  // calculate int = float / int
50  void intDivideFloatInt(int useStatic)
51  {
52      cout << "*** Input a float value and an integer value to calculate integer division **
53      float a {floatIn()};
54      int b {intIn()};
55      int result {0};         // Integer result
56      if (useStatic == 1)
57          result = a / static_cast<float>(b);
58      else if (useStatic == 2)
59          result = a / b;
60      cout << "Integer division result of float " << a << " / integer " << b << " is: ";
61      printResult(result, 0, true);
62  }
63
64  // calculate int = float / float
65  void intDivideFloatFloat()
66  {
67      cout << "*** Input two float values to calculate integer division ***\n";
68      float a {floatIn()};
69      float b {floatIn()};
70      int result {0};         // Integer result
71      result = a / b;
72      cout << "Integer division result of float " << a << " / float " << b << " is: ";
73      printResult(result, 0, true);
74  }
75
76  // calculate float = int / int
77  void floatDivideIntInt(int useStatic)
78  {
```

```
    cout << "*** Input two integer values to calculate float division ***\n";
    int a {intIn()};
    int b {intIn()};
    float result {0.0f};        // Float result
    if (useStatic == 1)
        result = static_cast<float>(a) / static_cast<float>(b);
    else if (useStatic == 2)
        result = a / b;
    cout << "Float division result of integer " << a << " / integer " << b << " is: ";
    printResult(0, result, false);
}

// calculate float = float / int
void floatDivideFloatInt(int useStatic)
{
    cout << "*** Input a float value and an integer value to calculate float division ***\
    float a {floatIn()};
    int b {intIn()};
    float result {0.0f};        // Float result
    if (useStatic == 1)
        result = a / static_cast<float>(b);
    else if (useStatic == 2)
        result = a / b;
    cout << "Float division result of float " << a << " / integer " << b << " is: ";
    printResult(0, result, false);
}

// calculate float = float / float
void floatDivideFloatFloat()
{
    cout << "*** Input two float values to calculate float division ***\n";
    float a {floatIn()};
    float b {floatIn()};
    float result {0.0f};        // Float result
    result = a / b;
    cout << "Float division result of float " << a << " / float "  << b << " is: ";
    printResult(0, result, false);
}

// * * * MAIN FUNCTION IS HERE * * *
int main()
{
    cout << "Use static_cast to convert integer inputs to float?\n"
            " * 1 - Yes\t*\n"
            " * 2 - No\t*\n";
    int useStatic {0};
    cin >> useStatic;
    if ( (useStatic != 1) && (useStatic != 2) ) {
        cout << "Invalid choice, the program will now terminate.\n";
        return 0;
    }

    cout << "Choose a calculation format:\n"
        " * 1 - int = int / int\t\t*\n"
        " * 2 - int = float / int\t*\n"
        " * 3 - int = float / float\t*\n"
        " * 4 - float = int / int\t*\n"
        " * 5 - float = float / int\t*\n"
        " * 6 - float = float / float\t*\n";
    int calcChoice {0};
    cin >> calcChoice;

```

```
    if (calcChoice == 1)
        intDivideIntInt(useStatic);
    else if (calcChoice == 2)
        intDivideFloatInt(useStatic);
    else if (calcChoice == 3)
        intDivideFloatFloat(); // doesn't require conversion, no ints in function, no args
    else if (calcChoice == 4)
        floatDivideIntInt(useStatic);
    else if (calcChoice == 5)
        floatDivideFloatInt(useStatic);
    else if (calcChoice == 6)
        floatDivideFloatFloat(); // doesn't require conversion, no ints in function, no ar
    else {
        cout << "Invalid choice, the program will now terminate.\n";
        return 0;
    }

    //print if int inputs were converted to floats
    if (useStatic == 1)
        cout << "Values were calculated by converting int inputs (if applicable) to float.

    return 0;
}
```

I'm loving programming. Why didn't I start earlier??

OK, let's jump to the quiz now. =D

sw1ft
February 22, 2019 at 6:36 pm · Reply

Wouldn't it be better to use a bool for your useStatic variable that you're passing to these functions? I mean it only has 2 options right? true and false?

Dimitri
January 12, 2019 at 5:14 am · Reply

my quiz#2

```
#include <iostream>
#include <limits>

int userInput()
{
    std::cout << "Enter an integer" << "\n";
    int x;
    std::cin >> x;
    return x;
}

bool isEven(int x)
{
    return (x %= 2) == 0;
}

void printResult(bool y)
{

    if (y)
        std::cout << "even" << "\n";
```

```
    else
        std::cout << "not even" << "\n";
}

int main()
{
    int x = userInput();
    bool y = isEven(x);

    printResult(y);

    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    std::cin.get();

    return 0;
}
```

**nascardriver**
January 12, 2019 at 5:22 am · Reply

Hi Dimitri!

* Line 28, 29: Initialize your variables with uniform initialization. You used copy initialization.
* Line 7: Initialize your variables with uniform initialization.
* Line 14: You don't need to modify @x

Dimitri
January 12, 2019 at 7:05 am · Reply

Thanks nascardriver! Got it! Except * Line 7: Initialize your variables with uniform initialization

Why we need initialization there?

**nascardriver**
January 12, 2019 at 7:10 am · Reply

Assuming you're using C++11 or later, it doesn't make a difference here. Initializing all variables will prevent you from forgetting an initialization where it's necessary.

Dimitri
January 12, 2019 at 7:26 am · Reply

Oh, OK! Better to assign 1 by default or it doesn't matter?

```
#include <iostream>
#include <limits>

int userInput()
{
    std::cout << "Enter an integer" << "\n";
    int x{ 1 };
    std::cin >> x;
    return x;
```

```
}

bool isEven(int x)
{
    return (x % 2) == 0;
}

void printResult(bool y)
{

    if (y)
        std::cout << "is even" << "\n";
    else
        std::cout << "is not even" << "\n";
}

int main()
{
    int x{ userInput() };
    bool y{ isEven(x) };

    printResult(y);

    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    std::cin.get();

    return 0;
}
```

**nascardriver**
January 12, 2019 at 7:28 am · Reply

Whatever is the 0 value for your type.
int 0
float 0.0f
double 0.0
etc

Dimitri
January 12, 2019 at 7:30 am · Reply

Many thanks!

Oscar R.
January 4, 2019 at 6:19 am · Reply

Hi Alex,
I have a question.

```
        // if count is evenly divisible by 20, print a new line
        if (count % 20 == 0)
            std::cout << "\n";
```

Shouldn't this got to the new line every 2 lines? 2 is evenly divisible by 20 and returns the remainder 0. Why does it still print every 20 lines?

**nascardriver**
January 4, 2019 at 6:26 am · Reply

You've got some pretty fancy math. 2/20 = 0,1

Oscar R.
January 4, 2019 at 6:39 am · Reply

*facepalm* I just realized. its late at night and im not thinking straight... sorry xd

**Shxbl**
January 13, 2019 at 12:05 am · Reply

2/20 = 0,1

0,1 * 20 = 2

is this  / symbol  means Division assignment ? @nascardriver

**nascardriver**
January 13, 2019 at 7:04 am · Reply

/ division
* multiplication
+ addition
- subtraction

Benjamin
January 3, 2019 at 5:14 pm · Reply

In the isEven() function i can up with:

```
1 bool isEven(int num){
2     return (num + 1) % 2;
3 }
```

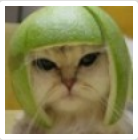
I read the comments and did not find anyone who did it this way. Is this ok as well or is their a problem with it?

**nascardriver**
January 4, 2019 at 4:59 am · Reply

Hi Benjamin!
Logically, your function is equivalent to the other submissions.
But your function is less efficient, because your computer has to perform the extra step of adding 1 to @num.

Benjamin
January 4, 2019 at 10:39 pm · Reply

Thank you! Wow i made two errors in my post. I meant "came* up with" and "or is there*". It was late at night, lol. Again thank you!



Alex
January 6, 2019 at 1:24 pm · Reply

While your function mostly works, it will overflow if you pass in the largest integer.

**nascardriver**
January 7, 2019 at 7:14 am · Reply

I had the same though, but I came to the conclusion that the overflow doesn't cause any problems, so I omitted it in my reply.
Assuming 32bit integers, 2147483647 (odd) will overflow to -2147483648 (even), same for all integer widths.
The only thing I can think of is that handling the overflow could take up a neglectable amount of CPU-time.

Qais
December 30, 2018 at 10:32 am · Reply

// ConsoleApplication4.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

```
#include "pch.h"
#include <iostream>

using namespace std;

int getNumber()
{
   int x;
   cout << "Please enter any number to see if it is odd or even:";
   cin >> x;
   return x;
}

bool calculate(int x)
{
   if (x % 2 == 0)
      return true;
   else
      return false;
}

int main()
{
   int x = getNumber();
   bool y = calculate(x);

   if (y == true)
      cout << "its even number.";
   else
      cout << "Its odd number.";
```

```
    return 0;
}
```

My program works fine but i am using different approach.

Can someone please clarify if my approach is fine?

**nascardriver**
December 31, 2018 at 5:41 am · Reply

Hi Qais!

* Initialize your variables with uniform initialization.
* In @calculate, you're returning the same value as the == comparison, so you might as well return it directly.
* Don't compare booleans to false/true, they're booleans already.
* Variable names should be descriptive.

Bad_at_Coding
December 18, 2018 at 12:59 pm · Reply

Hello there, even my code is working I don't know where should I improve it.So i hope i get some reply.
ps: about even or not even thing in printResult(),i dont know how it works(i guess it evaluates the first parametar if true, and the one after ":" if false), im new to programing, i just saw nascardriver use it in comments  some tutorials before and i kinda steal that :)

```
#include <iostream>

int getUserNumber()
{
    std::cout<<"Enter a number to check\n";
    int input { 0 };
    std::cin>>input;
    return input;
}
bool isEven(int user_input)
{
    return (user_input % 2 == 0);
}
void printResult(bool user_number_checked)
{
    std::cout<< "You number is "<< (user_number_checked ? "even": "not even");
}
int main()
{
    int user_number { getUserNumber() };
    bool user_number_checked { isEven(user_number) };
    printResult(user_number_checked);
    return 0;
}
```

**nascardriver**
December 19, 2018 at 4:09 am · Reply

Hi!

> I don't know where should I improve it
Looks fine to me :-)

> about even or not even thing in printResult [...]
That's the conditional operator. It's covered in lesson 3.4. Your description of how it works is correct.

Bad_at_Coding
December 20, 2018 at 4:29 am · Reply

Thanks for the replay nascardriver.Its very rare experienced programmer taking time to comment a begginer's simple programs.Its a kinda dying breed:)

GK
December 23, 2018 at 7:04 pm · Reply

Looks good, but some suggestions about general practices(more like being nit-picky :))

```
1 int input { 0 };
2 std::cin >> input;
```

>> Don't(more like 'No need to') initialize the variables if you are going to override them without using the initialized values.

```
1 void printResult(bool user_number_checked);
```

>> Input argument can be renamed to make it a bit more clear, something like 'is_num_even'

David
November 20, 2018 at 11:39 pm · Reply

Hi Alex

Firstly, huge thanks for this tutorial series. You have a great teaching style, and I really like how you take the time to talk about good programming practices and pitfalls.

Regarding the following code:

```
1 bool isEven(int x)
2 {
3     return (x % 2) == 0;
4 }
```

I was playing around with debug->disassembly in VS2017 and noticed that this actually compiles to more instructions in assembly than the if/else version, despite it seeming shorter/simpler in C++.

I didn't check the cycles per instruction, but either way, I thought this tutorial might be a good opportunity for you to bring up the topic of premature optimisation versus code readability (especially as I understand modern compilers can optimise better than most humans, and due to modern CPU features like speculative execution).

I'm just a novice/hobbyist programmer, so I used to be a bit OCD about trying to write efficient code (or at least code that I thought was efficient), including avoiding function calls due to their overhead, but am trying now to re-train myself to prioritise code readability. Personally I think the if/else version is just a tiny bit easier to understand intuitively at a glance.

Thanks again for your work on these tutorials!

**nascardriver**

November 21, 2018 at 5:15 am · Reply

Hi David!

Assembly has a very limited instruction set, whereas C++ is huge. You'll get a lot more instructions out of your code, no matter what you write.
Low-level optimization should be left to the compiler in most cases. The programmer should prioritize optimization of algorithms, data usage, etc., because those can't be optimized by the compiler and they have a much bigger impact than a couple of unnecessary instructions.

David
November 21, 2018 at 6:35 am · Reply

Yep! I got a little bit of exposure to ARM assembly a while back due to trying to mod some GBA/NDS games. It was fun and I learned a lot, but man is it good to be back in a higher level language and not have to keep track of things like the stack pointer for local variables!!

Agree with your points! I didn't really appreciate that when I started programming though, which is why I thought it might be a good topic for Alex to address in these tutorials as part of good programming practices. Before I was exposed to assembly, I used to think that less C++ code meant less instructions to execute - now I know that is not necessarily the case (for the isEven() example, both versions actually compile into the same assembly for Release build - just 3 instructions)! Now I also know that less instructions does not necessarily equal better performance (or matter nearly as much as algorithms and data usage, as you say).

Hence when it comes to choosing between if/else blocks versus code that looks shorter in C++, I now try to prioritise readability. It might be different for seasoned programmers like yourself, but I have to mentally convert that single line return statement in my head into an if/else statement when reading it anyway, so the if/else version just reads a tiny bit easier for me. The following code from Chapter 3.5 puts this in starker contrast. Also harder to debug I think? Breaking it out into if/else statements made it easier to follow in the debugger for me.

```
1 bool approximatelyEqual(double a, double b, double epsilon)
2 {
3     return fabs(a - b) <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
4 }
```

**nascardriver**
November 21, 2018 at 6:39 am · Reply

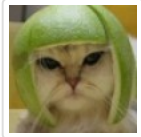
> Also harder to debug
Generally, if you use more lines, it's easier to debug. Using more lines doesn't necessarily change the compiled code.
@approximatelyEqual can be expanded into multiple lines without affecting the outcome. The one-liners might be easier written if you've been coding for a while. But once something goes wrong you'll be back to if/else and temporary variables.

You can edit your posts. The syntax highlighter will work after refreshing the page.

David
November 21, 2018 at 6:38 pm · Reply

> You can edit your posts. The syntax highlighter will work after refreshing the page.

Ha thanks! That's exactly why I deleted/re-posted. Hope you didn't have to retype your message because of that!



Alex
November 21, 2018 at 1:55 pm · Reply

Great conversational points, and the reminder not to prematurely optimize your code is timely as I'm rewriting a lesson where a mention of that would be perfect.

I agree the approximatelyEqual() function is hard to read/follow and even harder to debug -- but I opted to prefer the terse version under the assumption that most readers would just use the function, not try to dissect it. :) Maybe that's the wrong call to make in a learning tutorial series...

David
November 21, 2018 at 6:30 pm · Reply

> I'm rewriting a lesson where a mention of that would be perfect

Awesome! Is there a way to get email alerts when one of the lessons I've already worked through gets updated (and maybe also see what changed)? Couldn't find a subscribe button anywhere.

> I opted to prefer the terse version under the assumption that most readers would just use the function

Haha I think maybe you also chose it because you knew if you used the long version, you'd have to put up with lots of clever people telling you "did you know you could do that all in one line?" :p I could follow the logic in my head, but needed the debugger to figure out what was happening to the adjusted epsilon for the close-to-zero case. (Who knew multiplying a number by something <1 made that number smaller haha - my maths is sooo rusty!)

On the topic of debugging, I remember a few years ago when I first worked through these lessons (got up to Chapter 14) that the debugger was barely mentioned again after introducing it in Chapter 1 (from a quick site search it seems this is still the case). So back then I never really used the debugger - mostly due to lack of exposure and because I was used to just inserting "debug code" (eg, tracking how variables changed by printing to console), which is a habit I developed from modding games using LUA scripting. But after trying to make sense of ARM disassembly... my God how I appreciate a good debugger now!

As you update lessons for other issues, perhaps it would be helpful to start sprinkling in more exercises using the debugger (just to get new programmers more used to it). For example, in this lesson, the int pow() "exponentiation by squaring" algorithm causes an infinite loop if a negative exponent is entered. You could mention this and encourage students to use the debugger to figure out why (apparently depends on the compiler, but VS17 uses arithmetic shift right for signed int so the loop gets stuck at -1).

(I know you encourage robust code in later chapters, but I used to be super impatient and hate thinking about edge cases or investing the time/lines required. Until I started writing code to parse documents, and kept creating documents that broke my own code or which accidentally created infinite loops. Sometimes ya just gotta learn the hard way...)

Alex

November 27, 2018 at 12:41 pm · Reply

There currently isn't an email alert system for lesson updates. It's been a long-standing request that I haven't yet had time to figure out how to facilitate.

> Haha I think maybe you also chose it because you knew if you used the long version, you'd have to put up with lots of clever people telling you "did you know you could do that all in one line?" :p

I didn't think of that, but it's absolutely true. :)

Adding "debug this" quiz questions is absolutely on the agenda as I rewrite. I share your feeling that learning to use the debugger effectively is a crucial skill, and this tutorial series doesn't do a good enough job reinforcing this point.

I appreciate all the feedback! If you have any other thoughts, please feel free to share them.

David
November 27, 2018 at 8:06 pm · Reply

Hmm, the following WordPress plugins look like they might help? I don't really know much about WordPress or web development though...
https://wordpress.org/plugins/content-update-notification/
https://www.wpbeginner.com/plugins/how-to-show-post-updatesrevisions-to-your-readers-in-wordpress/

Alternatively, there is always the super low-tech but low-cost method: simply create a page to act as a manual changelog. Then from now on, whenever you update a lesson (other than typo fixes, etc) all you need to do is remember to add a one line summary to the changelog page with the date and lesson number.

> this tutorial series doesn't do a good enough job reinforcing this point

Think you're being a little bit hard on your own work there :)

There are a lot of important skills for larger projects or professional development (eg, version control, unit testing, different workflows, automating, etc) that this tutorial series couldn't possibly cover in a meaningful way without overwhelming readers and distracting from its core lessons. (I just spent a WEEK going down a Git rabbit-hole because it's integrated in Visual Studio, and that was just to learn the basics!)

What this tutorial series does do though (ie, cover fundamental C++ programming concepts step-by-step, while teaching good practices and how to avoid common pitfalls along the way), it does really REALLY well. The first time I came across this tutorial series, I had never encountered OOP, overloading, virtual functions and templates before. But this tutorial series made it a real joy to learn!

So thank YOU for all your time and effort!

Louis Cloete
January 9, 2019 at 4:13 pm · Reply

Since I wanted to write a comment about the int pow() function anyway and I wholeheartedly agree with David about the absolute brilliance (excellence? I don't know what word to use, but it should mean "wayyy better than anything else I have seen") of this website, I will add my 2c here to this thread.

First, let me say that this website has been a Godsend. I am a wannabe musician, and I use MuseScore, an open-source music notation editor written in C++ and using the Qt framework. I wanted to learn to code in C++ to contribute. I have learnt a little programming (Delphi, which is a Pascal framework, and the graphical language Scratch) in high school as an additional subject after school, but I dropped out because I wanted to spend more time with my music. Now I have to brush off my extremely small knowledge of programming and learn a new language.

I just have to say, Alex, you're an awesome teacher. You know in which order to treat subjects so that it makes logical sense, you have a gift for explaining and you know the value of lots of simple, easy-to-understand examples. And that is not all, you also provide very insightful comments into why certain programming practices are bad and why you should avoid them, even if the language allows it. Textbooks never even mention programming best practices. That is the difference it makes when the teacher is working in the industry every day. You know what works and what not. Professors don't necessarily know.

Now, to my comment about the int pow() function:

changing the parameters to

```
1 int pow(int base, unsigned int exp)
```

will sort out the problems with the undefined behaviour. You say yourself in the tutorial about bitwise operators that unsigned integers should always be used when doing bitwise operations. It will not violate your rule that you shouldn't mix signed and unsigned integers, since the exponent is never used in an expression with a signed variable.

David
January 9, 2019 at 4:45 pm · Reply

Hey Louis! Glad to see another huge fan of this website.

Re using an unsigned int parameter for int pow(), just be careful about C++ implicit conversion from int to unsigned int. I know because I previously tried to do exactly what you did, and was surprised that I could pass -1 to exp without any error/warning from the compiler. Not what you want because -1 becomes a very very large unsigned int!

I haven't tried this yet but apparently you can use templates (which is introduced in Chapter 13) to prevent such implicit conversions (see https://stackoverflow.com/a/44412113). If it works, would save on having to do error handling for exp < 0.

Louis Cloete
January 9, 2019 at 5:06 pm · Reply

Well, yeah, of course C++ would implicitly convert the int -1 to an unsigned int, but I figured it is better to let the loop iterate for a very large finite amount of iterations than to have an infinite loop.

Using unsigned int has the added benefit of telling the programmer that the function expects a positive number. The comment will never be seen if you put the function into a library and write a header file for it.

Maybe the best solution would be to figure out what is the largest exponent you can give the function without overflowing the int return value and test to see if your unsigned int is smaller than that. You should just use sizeof(int) to figure out if you are dealing with 16-bit or 32-bit ints.

I didn't read the stackoverflow thread yet when I wrote this. I'll have a look, but I did not get to templates yet. I restarted reading the tutorials about at the new year, because I stopped reading for quite some time and had to recap. I never read further that Chapter 9's comprehensive quiz.

David
January 9, 2019 at 5:44 pm · Reply

> Well, yeah, of course C++ would implicitly convert the int -1 to an unsigned int...

It wasn't obvious to me!! But yeah, infinite loops suck. One good thing about them though is it alerts you to a problem, whereas an incorrect result might go undetected - probably doesn't matter for the kind of things we're programming, but could be disastrous in something like aircraft software!

Templates (and virtual functions) are pretty fun when you get to them, once you wrap your head around the syntax. It's basically like a souped up version of overloading, but the compiler does some of the hard work for you.

Alex
January 10, 2019 at 10:05 am · Reply

Thanks for the kind words!

Using an unsigned parameter violates the "don't use unsigned integers except for bit-level operations", and also the "don't use unsigned values just to avoid negative numbers" rule (not sure if I've made that one explicit, I'll check and make sure I have).

The correct solution here is to use a signed parameter, but then use an assert or other precondition (in C++20, expects) to validate that the value actually is negative.

As David correctly notes, you'll still get conversions if you pass in a signed value as the parameter.

Louis Cloete
January 10, 2019 at 7:27 pm · Reply

@Alex, while I agree with your rules, I disagree that using an unsigned int is breaking them. Here is the function so you don't have to scroll up to see it:

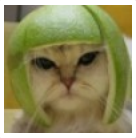
```
1 // note: exp must be non-negative
```

```
int pow(int base, int exp)
{
    int result = 1;
    while (exp)
    {
        if (exp & 1)
            result *= base;
        exp >>= 1;
        base *= base;
    }

    return result;
}
```

and here is my proposed change with clarifying comments:

```
int pow(int base, unsigned int exp)
{
    int result = 1;

    // No problem using unsigned int here. A zero-value will
    // everything else will be true. The exponent will be sh
    // until all the 1s are shifted off and the value is 0x0
    // This is actually expected behaviour.
    while (exp)
    {
        // This is actually a bit-level operation. Using a s
        // would actually cause undefined behaviour.
        if (exp & 1)

            // exp is not mixed with signed ints.
            // Here both operands are signed ints.
            result *= base;

        // This is a bit-level operation too. Using signed w
        // cause undefined behaviour (which is exactly what
        // David and why his program went into an infinite l
        exp >>= 1;

        // Here, yet again no mixing of signed and unsigned.
        base *= base;
    }

    // Return a signed int. Function has return type of sign
    // all is well, no implicit conversions.
    return result;
}
```

You would still need to assert the exp argument is not negative, but even if you do that, an unsigned int is better because you are using it for bit-level operations. Else you might anyway get undefined behaviour, even if you ensure that exp >= 0.



Alex
January 12, 2019 at 10:56 pm · Reply

We normally would choose unsigned for bit-level manipulation because we don't want surprises from the sign-bit. But:
1) Such surprises can't happen in this case -- if exp must be positive

(which we can enforce via assert or templatization), then the sign bit should always be 0, and we only do right shifts, so the sign bit never has a chance to flip.
2) We're treating this value as a number, not a sequence of bits. The bit manipulation here is an optimization, not a necessity (we could have used arithmetic instead, it just would have been slower).

Given the above, I'd still favor signed because then we can detect/handle the case when the caller passes in a negative value (even though the sample code doesn't, for simplicity), whereas with unsigned we can't (and we get a silent failure -- the worst kind).

Louis Cloete
January 13, 2019 at 1:49 pm

Ok, so do I understand correctly that bitshifts on signed positive integers (sign bit = 0) is defined? It is just the bitshifts with negative integers which exhibit undefined behaviour. (Some compilers pad with 0s and other compilers pad with 1s when you shift a signed int with a 1 in the sign bit to the right?)

Also, is bit operations with positive signed integers defined by the C++ language standard, or is it just a case of everybody agrees on how they should work, even if it isn't defined?

Alex
January 14, 2019 at 10:40 pm

Yes, I believe bit operations and binary representation for positive integers (and zero) are well defined by the C++ specification. It's only negative numbers that have issues, due to the fact that they can be encoded in different ways.

Louis Cloete
January 15, 2019 at 3:10 pm

You are correct. I looked it up here: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf It is in Chapter 5.8.

Bitshifts are defined by the standard for unsigned ints and non-negative signed ints as long as the result can be represented in the left operand's data type. For negative ints, the result is implementation-defined.

Rai
November 13, 2018 at 6:57 am · Reply

So I gave myself a challenge to write a geometric sequence and then display the last number in the sequence and state whether it's even or odd.

```cpp
#include <iostream>
#include <cmath>

double getNumber()
{
    std::cout << "Enter a number: " << std::endl;
    double number;
    std::cin >> number;
    return number;
}

int getPower()
{
    std::cout << "Enter a power: " << std::endl;
    int power;
    std::cin >> power;
    return power;
}

bool isEven(int x)
{
    return x % 2 == 0;
}

int main()
{
    double number = getNumber();
    int power = getPower();
    std::cout << "\nThe sequence\n____________\n" << number << std::endl;   // nothing impo

    double x = std::pow(number, power);
    while (x <= 1000000)
    {
        std::cout << x << " ";

        if ( static_cast<int>(x) % 4 == 0)   //useless but wanted to test it out.
            std::cout << "\n";

        x = std::pow(x, power);
    }

    if (isEven( static_cast<int>(x) ))
        std::cout << "\nYour last number " << x << " is even." << std::endl;
    else
        std::cout << "\nYour last number " << x << " is odd." << std::endl;
    return 0;
}
```

1. Is there any way I can improve this

2. I'm beginning to understand while loops. How do I do this:
if the user inputs a decimal number for "Enter a power: ". It'll output an speech and loop them back to question "Enter a power: " again until they give an integer without decimal point.

**nascardriver**
November 13, 2018 at 8:57 am · Reply

Hi Rai!

1.
* Initialize your variables with uniform initialization

* Line 32: You can use single quotation marks to make your numbers more readable (1'000'000)
* Line 32: This loop could be infinite if @power <= 1
* Use double numbers when calculating with doubles (1.0 instead of 1 etc.)

2.

```
bool hasFractionalPart(double db)
{
  // @std::floor removes the fractional part
  return (std::floor(db) != db);
  // If you don't know about flooring, you might be tempted to do something like this
  return (static_cast<double>(static_cast<int>(db)) != db);
  // But you'll run into problems where the integral type cannot represent numbers as
  // high as a double can.
}
```

Rai
November 13, 2018 at 12:39 pm · Reply

Thanks. One thing. You told me about Initialize your variables with uniform initialization before but I don't understand what this means - how to use {} to initialize variables. Could you show an example.

**nascardriver**
November 14, 2018 at 3:29 am · Reply

```
...
double number{ getNumber() };
int power{ getPower() };
...
```

This is part of lesson 2.1. Unfortunately, Alex doesn't use uniform initialization himself in most of the lessons.

« Older Comments 1 2 3 4