

# Arithmetic

2024 Winter APS 105: Computer Fundamentals

Jon Eyolfson

Lecture 4

1.0.1

## We Can Perform Arithmetic in C

An **expression** is a combination of operands and operators which results in a single value, e.g.  $1 + 2$  or  $3$

## We Can Perform Arithmetic in C

An **expression** is a combination of operands and operators which results in a single value, e.g.  $1 + 2$  or  $3$

There's all the operators you would expect:  $+$   $-$   $*$   $/$

They also follow the order of operations: BEDMAS

- Brackets

- Exponents

- Division / Multiplication

- Addition / Subtraction

We also work left to right, this is called **left-associative**

## It's Important to Agree on the Order

$$1 + 2 + 3$$

It's left-associative that means we compute it as  $(1 + 2) + 3$

If instead it was right-associate we compute it as  $1 + (2 + 3)$

$$1 + 2 * 3$$

Because  $*$  has higher precedence we compute it as  $1 + (2 * 3)$

All precedence rules are here: [CPlusPlus.com](http://Cplusplus.com)

## Division of `int` Values Behave Differently

If we write out the types of the operands, `int / int` results in another `int` value

If we compute `5 / 2` the result is `2`  
There's no rounding, the result is the quotient

## Division of `int` Values Behave Differently

If we write out the types of the operands, `int / int` results in another `int` value

If we compute `5 / 2` the result is `2`  
There's no rounding, the result is the quotient

We can use the `modulo` operator (`%`) to get the remainder  
(it has the same precedence as division and multiplication)

For example, the result of `5 % 2` is `1`

Both operands of a `%` must be `int`

## C Rules for Modulo Operator

The quotient is always **truncated**, which means:  
take what the result “should” be and chop off the decimal part

The rule for C is: if we can represent  $a / b$  then  
 $(a / b) * b + (a \% b)$  shall equal  $a$

Examples:

<b>a</b>	<b>b</b>	<b>a / b</b>	<b>a % b</b>
5	2	2	1
5	-2	-2	1
-5	2	-2	-1
-5	-2	2	-1

## Why Can't We Represent $a / b$ ?

The main thing we can't represent is: division by zero

In mathematics this is impossible, but we can write it in C

`1 / 0` will compile and give you a value



## Why Can't We Represent $a / b$ ?

The main thing we can't represent is: division by zero

In mathematics this is impossible, but we can write it in C

`1 / 0` will compile and give you a value

However, the value you get back is `undefined`

This means the value could be different every time,  
different on different computers, etc.

## Why Can't We Represent $a / b$ ?

The main thing we can't represent is: division by zero

In mathematics this is impossible, but we can write it in C

`1 / 0` will compile and give you a value

However, the value you get back is `undefined`

This means the value could be different every time,  
different on different computers, etc.

In C, we call this `undefined behavior` (US spelling)

Undefined behavior (UB) is one of the harder types of problems to debug

**You should avoid it at all costs!**

## What Happens If We Mix Types?

The operators `+` `-` `*` `/` `%` are all binary operators

If both operands are `int` the result is an `int`

Recall: for `%` both operands must be `int`

If at least one operand is a `double`, the other will be converted to a `double` if it is not one already

## When We Convert an `int` to a `double` We Add `.0`

If we write `2.5 + 2`, `2` (`int`) gets converted to `2.0` (`double`)  
The result will be `4.5`

This automatic type conversion is called `implicit type conversion`  
Implicit means we did not request the type conversion

## Conversion From a double to an int Truncates

We can do an **explicit type conversion** with a **type cast**

A type cast is a unary operator that has the form: (<type>) <value>

It has a higher precedence than the binary operators

It is right associative (right-to-left)

## Conversion From a double to an int Truncates

We can do an **explicit type conversion** with a **type cast**

A type cast is a unary operator that has the form: (<type>) <value>

It has a higher precedence than the binary operators

It is right associative (right-to-left)

(int) 2.9 gets truncated to 2

## Conversion From a double to an int Truncates

We can do an **explicit type conversion** with a **type cast**

A type cast is a unary operator that has the form: `(<type>) <value>`

It has a higher precedence than the binary operators

It is right associative (right-to-left)

`(int) 2.9` gets truncated to `2`

`(double) 5 / 2` gets computed as `((double) 5) / 2`

So, after the first step we get `5.0 / 2` then `2.5`

## Conversion From a double to an int Truncates

We can do an **explicit type conversion** with a **type cast**

A type cast is a unary operator that has the form: `(<type>) <value>`

It has a higher precedence than the binary operators

It is right associative (right-to-left)

`(int) 2.9` gets truncated to `2`

`(double) 5 / 2` gets computed as `((double) 5) / 2`

So, after the first step we get `5.0 / 2` then `2.5`

`(double) (int) 2.9` gets computed as `((double) ((int) 2.9))`

So, after the first step we get `(double) 2` then `2.0`



## Assignment is Also an Operator

It's a binary operator that's right associative,  
with the lowest precedence we've seen so far

The result of the assignment operator is the value assigned  
e.g. the result of `x = 3` is `3`

## Assignment is Also an Operator

It's a binary operator that's right associative,  
with the lowest precedence we've seen so far

The result of the assignment operator is the value assigned  
e.g. the result of  $x = 3$  is 3

This means  $y = x = 3$  gets computed as  $(y = (x = 3))$   
So, after the first step we assign 3 to x and we get  $y = (3)$   
then we assign 3 to y

## There's Other Shorthand Assignment Operators

You may find yourself doing something like:

```
x = x * 2;
```

Instead of this you can write:

```
x *= 2;
```

It applies the operation to the value on the left of the assignment, then re-assigns it with the result

## There's Other Shorthand Assignment Operators

You may find yourself doing something like:

```
x = x * 2;
```

Instead of this you can write:

```
x *= 2;
```

It applies the operation to the value on the left of the assignment, then re-assigns it with the result

There are shorthands for all the binary operators:

```
+= -= *= /= %=
```

## There's More Shorthands For Adding and Subtracting by 1

This is a very common operation while programming

Adding 1 to a variable is called **incrementing**

Subtracting 1 from a variable is called **decrementing**

The unary operator is ++ (increment), and -- (decrement)

For each there's two versions, the operator can either come:

before the variable (prefix), or

after the variable (postfix)

## Prefix and Postfix Increment and Decrement Differ

The difference between the two is the result of the operation

Assume initially we have: `int x = 0;`

`++x` will add 1 to x and the result is the updated value

In this case the result of `++x` is 1

## Prefix and Postfix Increment and Decrement Differ

The difference between the two is the result of the operation

Assume initially we have: `int x = 0;`

`++x` will add 1 to x and the result is the updated value

In this case the result of `++x` is 1

`x++` will add 1 to x and the result is the original value

In this case the result of `x++` is 0

## Prefix and Postfix Increment and Decrement Differ

The difference between the two is the result of the operation

Assume initially we have: `int x = 0;`

`++x` will add 1 to x and the result is the updated value

In this case the result of `++x` is 1

`x++` will add 1 to x and the result is the original value

In this case the result of `x++` is 0

In both cases x gets re-assigned a value of 1

**You should always prefer prefix over postfix unless necessary**



## sizeof Tells You the Number of Bytes Used

sizeof is a unary operator that works with variables or types  
The result is the number bytes used as an integer

We can use this to verify the number of bytes used for some types:

The result of sizeof(int) is 4


The result of sizeof(double) is 8

The result of sizeof(char) is 1

The result of sizeof(bool) is 1

If we declare double x; then the result of sizeof(x) is 8

## Summary of the Precedence Rules for Today's Operators

Operator	Associativity	
++ -- (postfix)	Left-to-right	 <p>Higher Precedence</p> <p>Lower Precedence</p>
++ -- (prefix) (<type>) (cast) & (address-of) sizeof	Right-to-left	
* / %	Left-to-right	
+ -	Left-to-right	
= += -= *= /= %=	Right-to-left	
(assignments)		

## You Can Add Comments to Your Code

A **comment** is a note for you and others to read, the compiler ignores them

Any text between `/*` and `*/` is considered a comment

You can also use `//`, anything after is ignored (until a newline)

## You Can Add Comments to Your Code

A **comment** is a note for you and others to read, the compiler ignores them

Any text between `/*` and `*/` is considered a comment

You can also use `//`, anything after is ignored (until a newline)

```
#include <stdio.h>
#include <stdlib.h> /* I need this for EXIT_SUCCESS. */

/* The program will start here.
   It will finish when it hits return. */
int main(void) {
    printf("Hello world\n");
    return EXIT_SUCCESS;
}
```

## Be Careful with Truncated Integer Division

If you ask someone, what is  $1/2 + 1/2$ ?

The answer should be 1, what would it be in C?

## Be Careful with Truncated Integer Division

If you ask someone, what is  $1/2 + 1/2$ ?

The answer should be 1, what would it be in C?

Since  $/$  has higher precedence than  $+$ , we would compute this as:

$$(1/2) + (1/2)$$

## Be Careful with Truncated Integer Division

If you ask someone, what is  $1/2 + 1/2$ ?

The answer should be 1, what would it be in C?

Since  $/$  has higher precedence than  $+$ , we would compute this as:

$$(1/2) + (1/2)$$

The result of  $1/2$  is  $0$ , so we get:

$$0 + 0 \text{ which is } 0$$