Learning Objectives: Arithmetic Operators

- **Recognize the symbols for arithmetic operators**
- **Use the `printf()` command to print doubles and integers**
- **Demonstrate how to increment and decrement a variable**
- **Perform string concatenation**
- **Apply PEMDAS rules to arithmetic operations**

Addition

The Addition (+) Operator

The **addition** operator works as you would expect with numbers. Copy and paste the code below into the text editor on the left. Make sure your code is in between the `//add code below this line` and `//add code above this line` comments. Then click the `TRY IT` button to see what is outputted by the code.

```
cout << 7 + 3 << endl;
```

You can also add two variables together. Modify the code to look like what's below and then click the `TRY IT` button again.

```
int a = 7;
int b = 3;
cout << a + b << endl;
```

challenge

What happens if you:

- Make `a` of type double (e.g. `double a = 7.0;`)?
- Change `a` to `double a = 7.1;`?
- Make `b` a negative number (e.g. `int b = -3;`)?
- Make `b` an explicitly positive number (e.g. `int b = +3;`)?

important

IMPORTANT

You may have noticed that when you add an int of `3` to a double of `7.1` you get `10.1`. However, when you add an int of `3` to a double of `7.0`, you get `10` instead of `10.0`. This occurs because by default `cout` does not print zeros after a decimal point *unless* those zeros are enclosed by other non-zero digits.

Examples:
* `cout << 7 + 3.14;` prints `10.14`
* `cout << 7.0 + 3.00;` prints `10`
* `cout << 7.00 + 3.01400;` prints `10.014`

==**Note**== that when an `int` and a `double` are added together, the result will be a `double` because the program will take on the data type that is more *flexible*.

Printing Floating Point Numbers

cout

The `cout` command is considered to be *non-specific* because you can use the same syntax for all of your printing needs (e.g. `cout << 1;`, `cout << "Hello";`, and `cout << true;`). However, for printing certain numbers, it is not always clear if what's printed is an `int` or a `double` sometimes.

```
int a = 1;
double b = 1.0;
cout << a << endl;
cout << b << endl;
```

Even though you are printing a `double` of `1.0`, the system will disregard the decimal and the trailing zero when `cout` is used. There is another print command called `printf()` that also prints text to the console.

printf()

`printf()` originates from the C language and, unlike the `cout` command, it is considered to be *specific*. This means that you must specify what *type of data* you want to print before you can use the command successfully.

```
int a = 1;
double b = 1.0;
cout << a << endl;
cout << b << endl;
printf("%d \n", a);
printf("%f \n", b);
```

challenge

What happens if you:

- Remove the `\n` from `printf("%d \n", a);`?
- Replace `%d` with `%f` in `printf("%d \n", a);`?
- Replace `%f` with `%d` in `printf("%f \n", b);`?

---

important

IMPORTANT

When `printf()` is used, a *specifier* is needed in order to tell the system what type of data you want to print. The `%d` tells the system to print an integer and `%f` tells the system to print a floating point number. If you use an incorrect specifier, you will receive an error message. By default, floating point numbers contain six zeros after the decimal point if they are printed using `printf()`.

The `\n` in `printf()` is equivalent to `endl`. They both print a newline character. Removing the `\n` from `printf("%d \n", a);` will delete the newline character and cause the variables `a` and `b` to be printed side-by-side.

---

cout vs. printf()

Unless you want to be *specific* with how your data is printed, you should always default to using `cout`. Only use `printf()` when formatting is important.

Incrementing Variables

Incrementing Variables

Incrementing a variable means to increase the value of a variable by a set amount. The most common incrementation you will see is when a variable increments itself by the value of 1.

```
int a = 0;
a = a + 1;
cout << a << endl;
```

How to Read `a = a + 1`

The variable `a` appears twice on the same line of code. But each instance of `a` refers to something different.

.guides/img/Increment

The ++ and += Operators

Since incrementing is such a common task for programmers, many programming languages have developed a shorthand for `a = a + 1`. The result is `a++` which does the same thing as `a = a + 1`.

```
int a = 0;
int b = 0;
a = a + 1;
b++;
cout << a << endl;
cout << b << endl;
```

In the cases where you need to increment by a different number, you can specify it by using the `+=` operator. You can replace `b++;` with `b+=1;` in the code above and still get the same result.

challenge

What happens if you:

- Replace `b++` in the code above with `b+=2`?
- Replace `b++` in the code above with `b+=-1`?
- Replace `b++` in the code above with `b-=1`?

String Concatenation

String Concatenation

String concatenation is the act of combining two strings together. This is done with the `+` operator.

```
string a = "This is an ";
string b = "example string";
string c = a + b;
cout << c << endl;
```

challenge

What happens if you:

- Concatenate two strings without an extra space (e.g. remove the space after `an` in `string a = "This is an";`)?
- Use the `+=` operator instead of the `+` operator (e.g. `a+=b` instead of `a + b`)?
- Add `3` to a string (e.g. `string c = a + b + 3;`)?
- Add `"3"` to a string (e.g. `string c = a + b + "3";`)?

Subtraction

Subtraction

Copy the code below and `TRY IT`.

```
int a = 10;
int b = 3;
int c = a - b;
cout << c << endl;
```

challenge

What happens if you:

- Assign `b` to `-3`?
- Assign `c` to `a - -b`?
- Assign `b` to `3.1`?
- Change `b` to `bool b = true;`?

important

IMPORTANT

Did you notice that you were able to subtract a `bool` from an `int`? Recall that a `bool` of `true` is actually an integer of `1` and `false` is actually `0`. Thus, the system is able to add and subtract `bool`s and `int`s. In addition, assigning `b` which is of type `int` to `3.1` will force the variable to adopt the integer value of `3` instead. Remember that all `int`s disregard decimal places.

The `--` and `-=` Operators

Decrementing is the opposite of incrementing. Just like you can increment with `++`, you can decrement using `--`.

```
int a = 10;
a--;
cout << a << endl;
```

Like `+=`, there is a shorthand for decrementing a variable, `-=`. For example, if you want to decrement the variable `a` by `2` instead of `1`, replace `a--` with `a-=2`.

Subtraction and Strings

You might be able to concatenate strings with the `+` operator, but you cannot use the `-` operator with them.

```
string a = "one two three";
string b = "one";
string c = a - b;
cout << c << endl;
```

Division

Division

Division in C++ is done with the `/` operator.

```
double a = 25.0;
double b = 4.0;
printf("%f \n", a / b);
```

challenge

What happens if you:

- Assign `b` to `0.0`?
- Assign `b` to `0.5`?
- Change the code to...

```
double a = 25.0;
double b = 4.0;
a /= b;
printf("%f \n", a);
```

▼ **Hint(s)**

`/=` works similarly to `+=` and `-=`.

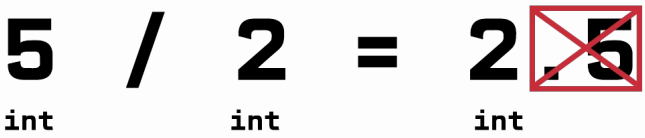
important

IMPORTANT

Division by zero is *undefined* in mathematics. In C++, dividing by an **integer** of `0` results in an error message. However, dividing by a **double** of `0.0` results in `inf` which is short for *infinity*.

Integer Division

Normally, you use `double` in C++ division since the result usually involves decimals. If you use integers, the division operator returns an `int`. This "integer division" does not round up, nor round down. It removes the decimal value from the answer.



.guides/img/IntDivision

```
int a = 5;
int b = 2;
cout << a / b << endl;
```

Modulo

Modulo

Modulo is the mathematical operation that performs division but returns the remainder. The modulo operator is `%`.

.guides/img/Modulo

```cpp
int modulo = 5 % 2;
cout << modulo << endl;
```

challenge

What happens if you:

- Assign `modulo` to `5 % -2`?
- Assign `modulo` to `5 % 0`?
- Assign `modulo` to `5 % 2.0`?

Multiplication

Multiplication

C++ uses the `*` operator for multiplication.

```
int a = 5;
int b = 10;
cout << a * b << endl;
```

challenge

What happens if you:

- Assign `b` to `0.1`?
- Assign `b` to `-3`?
- Change the code to...

```
int a = 5;
int b = 10;
a*=b;
cout << a << endl;
```

▼ **Hint(s)**

`*=` works similarly to `+=`, `-=`, and `/=`.

Order of Operations

Order of Operations

C++ uses the **PEMDAS** method for determining order of operations.

P Parentheses
E Exponents - powers & square roots
MD Multiplication & Division - left to right
AS Addition & Subtraction - left to right

.guides/img/PEMDAS

By default, there are no operators for **exponents** and **square roots**. Instead, functions such `pow( , )` and `sqrt()` are used to calculate powers and square roots respectively. In order to use these functions, they must be imported by including `#include <cmath>` at top of the program header. For exponents, the *base* number goes before the `,` in `pow( , )` and the *exponent* goes after the `,`. For example, `pow(4, 2)` calculates 4^2 and `pow(4, 0.5)` calculates $4^{0.5}$ or $4^{1/2}$. For square roots, the number goes inside the `()` in `sqrt()`. An example is `sqrt(4)` which calculates $\sqrt{4}$.

```
cout << pow(2, 2) << endl;
cout << pow(25, (1 / 2)) << endl;
cout << pow(25, (1.0 / 2.0)) << endl;
cout << sqrt(25) << endl;
```

▼ **pow(25, (1 / 2)) vs. pow(25, (1.0 / 2.0))**
`pow(25, (1 / 2))` results in `1` because integer division is performed within `(1 / 2)`. `1` divided by `2` returns in an integer of `0` and 25^0 computes to `1`. On the other hand, `pow(25, (1.0 / 2.0))` involves double division which is why `5` was computed.

The code below should output `10.000000`.

```
int a = 2;
int b = 3;
int c = 4;
double result = 3 * a - 2 / (b + 5) + c;
printf("%f \n", result);
```

▼ **Explanation**

- The first step is to compute `b + 5` (which is `8`) because it is surrounded by parentheses.
- Next, do the multiplication and division going from left to right: `3 * a` is `6`.
- `2` divided by `8` is `0` (remember, the `/` operator returns an `int` when you use two `int`s so `0.25` becomes `0`).
- Next, perform addition and subtraction from left to right: `6 - 0` is `6`.
- Finally, add `6` and `4` together to get `10`.
- Since `result` is of type `double`, `10.000000` is printed.

challenge

Mental Math

```
5 + 7 - 10 * 4 / 2
```

▼ Solution

-8

```
5 * 8 - 7 % 2 - 18 * -1
```

▼ Solution

57

```
9 / 3 + (100 % 100) - 3
```

▼ Solution

0

```
12 - 2 * pow(2, 3) / (4 + 4)
```

▼ Solution

10

Type Casting

Type Casting

Type casting (or type conversion) is when you change the data type of a variable.

```
int numerator = 40;
int denominator = 25;
int number = 0;
cout << boolalpha << (bool) number << endl;
cout << numerator / denominator << endl;
cout << (double) numerator / denominator << endl;
```

`numerator` and `denominator` are integers, but `(double)` converts `numerator` into a double. You can use `(double)`, `(int)`, and `(bool)` to cast any double, integer, or boolean between each other. Note that casting an integer of `0` or a double of `0.0` to a boolean will result in `false`. Any other integer or double values will result in `true`.

challenge

What happens if you:

- Assign `number` to `5`?
- Cast only `denominator` to a double?
- Cast both `numerator` and `denominator` to a double?
- Cast the result to a double (e.g. `(double) (numerator / denominator)`?
- Change the code to...

```
int numerator = 40;
int denominator = 25;
int number = 5;
cout << boolalpha << (bool) number << endl;
cout << numerator / denominator << endl;
cout << (double) numerator / denominator << endl;
printf("%d \n", numerator / denominator);
printf("%f \n", (double) numerator / denominator);
printf("%f \n", (double) (numerator / denominator));
```

▼ **More information**

If either or both numbers in C++ division are `doubles`, then `double` division will occur. In the last example, `numerator` and `denominator` are both `ints` when the division takes place which results in an `int` of `1`. An integer of `1` converted to a `double` is `1.000000` but `cout` removes the decimal point and all of the trailing zeros.

Data Type Compatibility

Do you know why the code below will not work?

```
int a = 5;
string b = "3";
cout << a + b << endl;
```

In C++, you can add a combination of `ints`, `doubles`, and `bools` together. Remember that a boolean is either `1` if it's `true` or `0` if it's `false`. In the example above, adding a string to an integer results in an error. That's because a `string` has no numerical value and can only be added to other `strings`. However, you can convert the string `b` to an integer to fix the

problem by using `stoi()`. `stoi()` acts as a **function** to convert a string into an integer. The string or string variable goes into the `()` to be converted. See below for a list of type-conversion functions.

```
int a = 5;
string b = "3";
string c = "3.14";
bool d = true;
cout << a + stoi(b) << endl;
```

▼ **List of commonly used type-conversion functions**

Function	Input Type	Output Type	Example
`stoi()`	string	int	`stoi("10")`
`stod()`	string	double	`stod("12.34")`
`to_string()`	int, double, or boolean	string	`to_string(10)`, `to_string(12.34)`, or `to_string(false)`

challenge

What happens if you:

- Replace `stoi(b)` with `stoi(c)`?
- Replace `stoi(b)` with `stod(c)`?
- Replace `stoi(b)` with `to_string(d)`?
- Replace `a + stoi(b)` with `b + to_string(d)`?

important

IMPORTANT

You can convert the string `"3.14"` to an integer using `stoi()` which will result in an int of `3`. To retain the decimal places, use `stod()` instead. In addition, the `to_string()` function will convert a boolean into the string form of its numerical value. `to_string(true)` will convert `true` to `"1"` instead of `1`. This is why adding `b`, which is a string of `"3"`, to `to_string(d)` resulted in the string of `"31"`.