Unit 7: Arithmetic Operations

Learning Objectives

After this unit, students should:

- be able to define arithmetic expressions in C programs that include the use of the following arithmetic operators for:
  - addition (`+`),
  - subtraction (`-`),
  - multiplication (`*`),
  - division (`/`), and
  - modulo (`%`);
- be aware of the numeric types that may be used with arithmetic operators in C programs;
- be aware of the value range restrictions on the various numeric types in C;
- be able to use compound operators in C programs;
- be aware of the difference between division and integer division, and when the latter occurs within arithmetic expressions; and
- be aware that there exist pitfalls in using `++` and `--` operators and they are not allowed in CS1010.

Operators

You have seen the `+` operator in the previous units. You can use `+` to add two variables, a value, and a variable, or two values:

```
long a = 1;
long b = 2;
long c = 3;
a = b + c; // add two variables
b = a + 4; // add a variable to a value
c = 5 + 6; // add two values
```

You can also use `+` on values returned by functions:

```
hypotenuse = sqrt(square(base) + square(height));
```

You have also seen the multiplication operator `*`. It can be used in the same way as the `+` operator. Three other useful operators are:

- `/` - division (e.g., `double half_x = x / 2;`)
- `-` - subtraction (e.g., `long deducted = income - 100`)
- `%` - modulo (e.g, `long last_digit = number % 10`);

The `+`, `-`, `*`, and `/` operators work on both integer types (`char`, `short`, `int`, `long`, `long long`) and real numbers (`float`, `double`). The module operator `%` works only on integer types.

Operator Precedence

We can chain the operations together to form expressions such as:

```
long b = 10;
long c = 2;
long a = b + 2 * c / 4;
```

When we have multiple operations appearing, however, it becomes harder to trace the sequence of evaluation. What is the value of `a` after

the three lines above are executed?

C actually has well-defined rules to the order of evaluation for the operators: `*`, `/`, and `%` take precedence over `+` and `-`, and the operators are evaluated from left to right.

Thus, in the example above, `a` will be 11 instead of 6 after the execution.

To change the order of execution, we can add parenthesis to the expression. For instance:

```
long a = (b + 2) * c / 4; // 6
long a = b + (2 * c / 4); // 11
```

The expression in the parenthesis will be evaluated first. To make your code easier to understand, *you should add parenthesis even if the order of evaluation is from left to right* to make the order of evaluation explicit.

Compound Operators

It is common to modify the value of a variable and store new value back to the same variable. For example,

```
index = index + 1;  // increment the variable index
age = age * 2;  // double the variable age
```

C provides *compound operators* that simplify the expressions above. For example,

```
index += 1;
age *= 2;
```

The syntax for a compound operator is `op=`, where `op` can be `+`, `-`, `*`, `/`, `%`, or other binary operators. The statement:

```
a op= b;
```

modifies `a` the same way as:

```
a = a op b
```

Common Mistakes Using Arithmetic Operations

It is important to remember that when arithmetic operations in C are performed on a sequence of bits, the value that the sequence of bits can represent is limited and is determined by its type. A common mistake for beginner programmers is to forget this fact and treat the arithmetic operations as the same as the ones seen in mathematics.

Let's look at two common gotchas.

Overflow

Consider the type `uint8_t`, which represents an unsigned 8-bit integer and the following code:

```
uint8_t c = 255;
c += 1;
```

What is the value of variable `c` after the operation above?

Here, we are adding one to the value 255, so `c` must store the value 256, right?

It turns out that after the execution above, `c` contains the value 0. The variable `c` is of the type `uint8_t`, which is the unsigned 8-bit integer.

Being 8-bit, the variable can store values from 0 to 255. When we add 1 to 255, even though we get the result 256, mathematically, we cannot store 256 in `c` -- there are not enough bits! In this case, the value stored is "wrap around", and we get the value 0 instead.

The variable `c` above is unsigned. It gets trickier if `c` is signed. In the case of overflowing signed integer, the behavior depends on the compiler and is undefined in the C standard.

Integer Division

Now, let's consider the following code:

```
1  double half = 3/2;
```

What is the value of variable `half` after the operation above?

It got to be 1.5, right?

It turns out that, after executing the code above, the value of `half` is 1.0. 😮

To understand this, first, let's see what happen when we assign a floating-point number to an integer type:

```
1  int x = 1.5;
```

C truncates the floating number and only stores the integer part of the value, 1 in this case, in `x`.

Second, when we perform an arithmetic operation, the resulting value will be an integer if both values are integer types. If one of the operands is a floating-point number, the result will be a floating-point number[1].

Since 3 and 2 are both integers, the resulting value 1.5 is stored in an integer, which causes it to become 1. We then store 1 into a `double` variable, causing the value of `half` to become `1.0`.

Because of this limitation, the operation `/` is sometimes also known as integer division when both operands are integers.

In order to get the result 1.5 as expected, we can write either:

```
1 double half = 3/2.0;
```

or

```
1 double half = 3/(double)2;
```

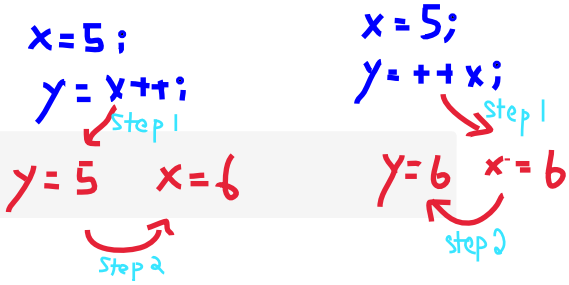
The second fix above explicitly converts the type, or *casts* the type of value 2 into a `double`.

Avoid Increment / Decrement Operator

If you read C code in other places, you will certainly come across the increment or decrement operator, `++` or `--`. The operators add one and minus one from the operand respectively. So, the statement

```
1 index += 1;
```

can be further shortened into




```
1 index++;
```

and the statement

```
1 index -= 1;
```

can be further shortened into

```
1  index--;
```

Using these two operators only shorten your code by two characters per statement, but introduces several issues. As such, we *ban the use of both increment and decrement operator in CS1010*.

So, why aren't `++` and `--` welcomed in CS1010? The `++` and `--` operators not only modify the value of the operand, but it also returns a value. We can write `j = i++;` to both increment `i` and assign the pre-incremented value of `i` to `j`. In C, we can also write `j = ++i;`, which again, increment `i`, and assign the post-incremented value of `i` to `j`. Things get tricky, when we write `i = i++;`, it is not clear how to interpret this. The C standard leaves this behavior undefined and leaves it to the compiler to define its behavior. Introducing all these complexities just to save two characters is not warranted.

List of C features banned in CS1010

You should realize by now that we are only using a subset of C and enforce a certain style of programming in CS1010. We wish to move all of you away from common pitfalls in learning and writing C so that we can focus on using C as a tool to solve problems.

So far, you have seen that we are banning: - `++` and `--` operators - the types `int`, `short`, `float` etc. (with the returning type of `main()` as the exception) - global variables

In addition, we discourage - the use of `printf` and `scanf` - skipping parenthesis in writing arithmetic expression

The complete list of C features that we ban or discourage in CS1010 can be found here.

1. The actual rules used by C, called *integer promotion* and *usual arithmetic conversion*, are much more complex and are outside the scope of CS1010. You should take note of this, however, and in a later part of your study or career, if you need to delve deeper into writing or debugging C code, take a look at this. ↵