

Arithmetic Expressions Lesson #2 Outline

1. Arithmetic Expressions Lesson #2 Outline
2. Named Constant & Variable Operands #1
3. Named Constant & Variable Operands #2
4. Named Constant & Variable Operands #3
5. Constant-Valued Expressions #1
6. Constant-Valued Expressions #2
7. Constant-Valued Expressions #3
8. Assignments W/O Expressions: Not Very Useful
9. Assignments with Expressions: Crucial
10. Meaning of Assignment w/Expression
11. Assignment w/Expression Example
12. Assignment w/Same Variable on Both Sides
13. Same Variable on Both Sides: Meaning
14. Same Variable on Both Sides: Example
15. Single Mode Arithmetic
16. `int` vs `float` Arithmetic
17. `int` vs `float` Division
18. `int` Division Truncates
19. Division By Zero
20. Division By Zero Example #1
21. Division By Zero Example #2
22. Floating Point Exception
23. Mixed Mode Arithmetic #1
24. Mixed Mode Arithmetic #2
25. Promoting an `int` to a `float`
26. Programming Exercise



Named Constant & Variable Operands #1

So far, many of the examples of expressions that we've looked at have used numeric literal constants as operands.

But of course we already know that using numeric literal constants in the body of a program is BAD BAD BAD.

So instead, we want to use named constants and variables as operands.



Named Constant & Variable Operands #2

```
#include <stdio.h>

int main ()
{ /* main */
    const int days_in_a_year      = 365;
    const int hours_in_a_day      = 24;
    const int minutes_in_an_hour  = 60;
    const int seconds_in_a_minute = 60;
    const int program_success_code = 0;
    int year_of_birth, current_year, age_in_seconds;

    printf("Let me guess your age in seconds!\n");
    printf("What year were you born?\n");
    scanf("%d", &year_of_birth);
    printf("What year is this?\n");
    scanf("%d", &current_year);
    age_in_seconds =
        (current_year - year_of_birth) *
        days_in_a_year * hours_in_a_day *
        minutes_in_an_hour * seconds_in_a_minute;
    printf("I'd guess that your age is about");
    printf(" %d seconds.\n", age_in_seconds);
    return program_success_code;
} /* main */
```



Named Constant & Variable Operands #3

```
% gcc -o age_in_seconds age_in_seconds.c
```

```
% age_in_seconds
```

```
Let me guess your age in seconds!
```

```
What year were you born?
```

```
1985
```

```
What year is this?
```

```
2024
```

```
I'd guess that your age is about 1229904000 seconds.
```



Constant-Valued Expressions #1

A *constant-valued expression* is an expression all of whose terms are constants (so its value, when evaluated, is also a constant).

If we have an expression whose terms are all constants (either literal constants or named constants), then we can use that expression in the initialization of a named constant:

```
const float C_to_F_factor          = 9.0 / 5.0;
const float C_to_F_increase       = 32.0;
const float C_water_boiling_temperature = 100.0;
const float F_water_boiling_temperature =
    C_water_boiling_temperature *
    C_to_F_factor + C_to_F_increase;
```



Constant-Valued Expressions #2

```
#include <stdio.h>
```

```
int main ()
{ /* main */
    const float C_to_F_factor          = 9.0 / 5.0;
    const float C_to_F_increase        = 32.0;
    const float C_water_boiling_temperature = 100.0;
    const float F_water_boiling_temperature =
        C_water_boiling_temperature *
        C_to_F_factor + C_to_F_increase;

    printf("Water boils at %f degrees C,\n",
        C_water_boiling_temperature);
    printf("          which is %f degrees F.\n",
        F_water_boiling_temperature);
} /* main */
```

NOTE: In the initialization of a named constant,
we **CANNOT** have an expression whose value is **NOT** a constant.



Constant-Valued Expressions #3

```
% gcc -o constant_expression constant_expression.c
```

```
% constant_expression
```

```
Water boils at 100.000000 degrees C,  
            which is 212.000000 degrees F.
```

NOTE: In the initialization of a named constant,
we **CANNOT** have an expression whose value is **NOT** a constant.



Assignments W/O Expressions: Not Very Useful

So far, many of the assignment statements that we've seen have simply assigned a literal value to a variable:

```
% cat variable_assignment.c
#include <stdio.h>

int main ()
{ /* main */
    int x;

    x = 5;
    printf("x = %d\n", x);
} /* main */
% gcc -o variable_assignment variable_assignment.c
% variable_assignment
x = 5
```

Unfortunately, this is not very interesting and won't accomplish much in an actual real life program.

To make a program useful, most of the assignments have to have expressions on the right hand side.



Assignments with Expressions: Crucial

```
% cat triangle_area.c
#include <stdio.h>

int main ()
{ /* main */
    const float height_factor = 0.5;
    float base, height, area;

    printf("This program calculates the area of a\n");
    printf("triangle from its base and height.\n");
    printf("What are the base and height?\n");
    scanf("%f %f", &base, &height);
    area = height_factor * base * height;
    printf("The area of a triangle of base %f\n", base);
    printf("and height %f is %f.\n", height, area);
} /* main */

% gcc -o triangle_area triangle_area.c
% triangle_area
This program calculates the area of a
triangle from its base and height.
What are the base and height?
5 7
The area of a triangle of base 5.000000
and height 7.000000 is 17.500000.
```



Meaning of Assignment w/Expression

Suppose that we have an expression on the right hand side of an assignment:

$$x = y + 1;$$

Remember:

An assignment statement is an **ACTION**, **NOT** an equation.

The compiler interprets this statement to mean:

- “first, **evaluate the expression** that’s on the right hand side of the assignment operator (single equals sign);
- then, **put the resulting value** into the variable that’s on the left side of the assignment operator (single equals sign).”

In the example above, the assignment statement means:

“evaluate $y + 1$, then put the resulting value into x .”



Assignment w/Expression Example

```
% cat x_gets_y_plus_1.c
#include <stdio.h>

int main ()
{ /* main */
    int x, y;

    y = 5;
    printf("y = %d\n", y);
    x = y + 1;
    printf("x = %d\n", x);
} /* main */

% gcc -o x_gets_y_plus_1 x_gets_y_plus_1.c
% x_gets_y_plus_1
y = 5
x = 6
```



Assignment w/Same Variable on Both Sides

Here's another assignment:

$$x = x + 1;$$

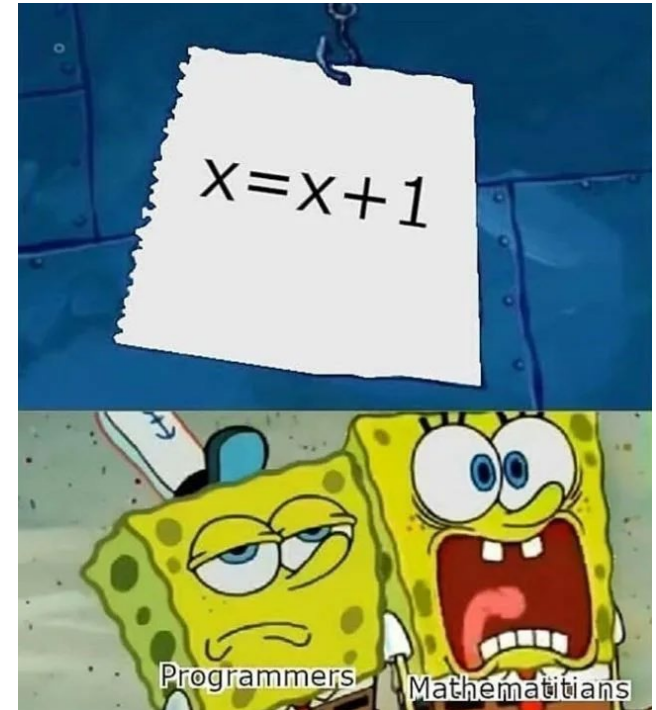
The assignment statement above might be confusing, because it has the same variable, x , on both the left hand side and the right hand side of the single equals sign.

**IF THIS WERE AN EQUATION,
IT'D BE BAD.**

But it's **NOT** an equation, it's an **ACTION**.

So the assignment above is **GOOD**.

https://img-9gag-fun.9cache.com/photo/av59v7X_700bwp.webp



Same Variable on Both Sides: Meaning

$$x = x + 1;$$

In general, the compiler interprets an assignment statement to mean:

- “first, **evaluate the expression** that’s on the right hand side of the assignment operator (equals sign);
- then, **put the resulting value** into the variable that’s on the left hand side of the assignment operator (equals sign).”

So, the assignment statement above means:

“Get the current value of x , then add 1 to it, then put the new value **back into** x , replacing the previous value.”



Same Variable on Both Sides: Example

```
% cat assign_self.c
#include <stdio.h>

int main ()
{ /* main */
    int x;

    x = 5;
    printf("After 1st assignment, x = %d\n", x);
    x = x + 1;
    printf("After 2nd assignment, x = %d\n", x);
} /* main */

% gcc -o assign_self assign_self.c
% assign_self
After 1st assignment, x = 5
After 2nd assignment, x = 6
```



Single Mode Arithmetic

In C, when we have an arithmetic expression whose terms all evaluate to a single data type (for example, all `int`-valued terms or all `float`-valued terms), we refer to this as *single mode arithmetic*.

In C, single mode `int` arithmetic behaves like single mode `float` arithmetic most of the time.



`int` vs `float` Arithmetic

In C, single mode `int` arithmetic behaves like single mode `float` arithmetic most of the time.

`5.0 + 7.0` is `12.0` and

`5 + 7` is `12`

`5.0 - 7.0` is `-2.0` and

`5 - 7` is `-2`

`5.0 * 7.0` is `35.0` and

`5 * 7` is `35`

But, division is different for `int` versus `float`!



`int` vs `float` Division

Division is different for `int` versus `float`!

<code>5.0</code>	<code>/</code>	<code>7.0</code>	is	<code>0.71...</code>	<u>BUT</u>
<code>5</code>	<code>/</code>	<code>7</code>	is	<code>0</code>	

We see that `float` division in C works the same way that division works in mathematics.

But `int` division is a little bit strange.

In `int` division, the result is **truncated** to the nearest `int` whose absolute value is immediately less than or equal to the mathematical result.

Truncate: to cut off (for example, to cut off the digits to the right of the decimal point)



int Division Truncates

4.0 / 4.0 is 1.0 and

4 / 4 is 1

5.0 / 4.0 is 1.25 **BUT**

5 / 4 is 1

6.0 / 4.0 is 1.5 **BUT**

6 / 4 is 1

7.0 / 4.0 is 1.75 **BUT**

7 / 4 is 1

8.0 / 4.0 is 2.0 and

8 / 4 is 2



Division By Zero

Mathematically, division by zero gives an **infinite result**:

$$\frac{c}{0} = \infty \text{ for } c \neq 0$$

Or, more accurately, if you've taken Calculus:

“The limit of c / x as x approaches zero is arbitrarily large.”

Computationally, division by zero causes an **error**.



Division By Zero Example #1

```
% cat divide_by_zero_constant.c
```

```
#include <stdio.h>
```

```
int main ()
```

```
{ /* main */
```

```
    printf("5 / 0 = %d\n", 5 / 0);
```

```
} /* main */
```

```
% gcc -o divide_by_zero_constant divide_by_zero_constant.c
```

```
divide_by_zero_constant.c: In function 'main':
```

```
divide_by_zero_constant.c:4: warning: division by zero
```



Division By Zero Example #2

```
% cat divide_by_zero.c
#include <stdio.h>

int main ()
{ /* main */
    int numerator, denominator, quotient;

    printf("What's the numerator?\n");
    scanf("%d", &numerator);
    printf("What's the denominator?\n");
    scanf("%d", &denominator);
    printf("numerator    = %d\n", numerator);
    printf("denominator = %d\n", denominator);
    quotient = numerator / denominator;
    printf("numerator / denominator = %d\n", quotient);
} /* main */

% gcc -o divide_by_zero divide_by_zero.c
% ./divide_by_zero
What's the numerator?
5
What's the denominator?
0
numerator    = 5
denominator = 0
Floating exception
```



Floating Point Exception

```
% gcc -o divide_by_zero divide_by_zero.c  
% divide_by_zero
```

What's the numerator?

5

What's the denominator?

0

numerator = 5

denominator = 0

Floating exception

Note that, in the context of computing, the word
exception means “a very dumb thing to do.”

As in, “I take exception to that.”



Mixed Mode Arithmetic #1

In principle, we might like our numeric expressions to have either all `int`-valued terms or all `float`-valued terms.

In practice, we can, and often must,

mix `int`-valued and `float`-valued terms

– literals, named constants, variables and subexpressions –
subject to the rule that an operation with
operands of both data types has a `float` result.

We call such expressions ***mixed mode*** arithmetic.



Mixed Mode Arithmetic #2

1 + 2 is 3 **BUT**

1.0 + 2 is 3.0 and

1 + 2.0 is 3.0

1 - 2 is -1 **BUT**

1.0 - 2 is -1.0 and

1 - 2.0 is -1.0

1 * 2 is 2 **BUT**

1.0 * 2 is 2.0 and

1 * 2.0 is 2.0

1 / 2 is 0 **BUT**

1.0 / 2 is 0.5 and

1 / 2.0 is 0.5



Promoting an `int` to a `float`

For mixed mode arithmetic, we say that
an `int` operand is *promoted* to `float`.

`1 / 2` is `0` **BUT**

`1 / 2.0` is

`1.0 / 2.0` is `0.5`

`4.0 / (3 / 2)` is `4.0` **BUT**

`4.0 / (3.0 / 2)` is

`4.0 / (3.0 / 2.0)` is `2.666...`



Programming Exercise

Given a weight/mass in pounds, convert to weight/mass in metric tons.

Specifically, draw a flowchart and then write a C program that:

1. greets the user;
2. prompts the user and then inputs an weight/mass in pounds;
3. calculates the weight/mass in metric tons;
4. outputs the weight/mass in both pounds and metric tons.

The body of the program must not have any numeric literal constants; all constants must be declared using appropriate identifiers.

Don't worry about comments.

