Arithmetic Expressions Outline

1. Arithmetic Expressions Outline
2. Expressions
3. Arithmetic Expressions
4. Unary & Binary Arithmetic Operations
5. Structure of Arithmetic Expressions/
   Jargon: int-valued & float-valued Expressions
6. Precedence Order of Arithmetic Operations
7. Arithmetic Precedence Order Example: int Operands
8. Arithmetic Precedence Order Example: float Operands
9. Named Constants & Variables as Operands
10. Constant-Valued Expressions in Named Constant Initializations
11. Assignments Using Expressions
12. Using the Same Variable on Both Sides of an Assignment
13. The Meaning of Using the Same Variable on Both Sides of an Assignment
14. Single Mode Arithmetic
15. Division By Zero
16. Mixed Mode Arithmetic
17. Exercise: Writing a Program
Expressions

\[ a + b - c \times d / e \% f - 398 + g \times 5981 / 15 \% h \]

In programming, an *expression* is a combination of:

- **Operands**, such as:
  - Literal constants
  - Named constants
  - Variables
  - *Function invocations* (which we’ll discuss later)

- **Operators**, such as:
  - Arithmetic Operators
    - Addition: +
    - Subtraction: –
    - Multiplication: *
    - Division: /
    - Modulus (remainder): % (only for int operands)
  - Relational Operators
    - Equal: ==
    - Not Equal: !=
    - Less Than: <
    - Less Than or Equal To: <=
    - Greater Than: >
    - Greater Than or Equal To: >=
  - Logical Operators
    - Negation: !
    - Conjunction (AND): &&
    - Disjunction (OR): ||

- Parentheses: (    )

Not surprisingly, an expression in a program can look very much like an expression in math (though not identical). This is on purpose.
Arithmetic Expressions

An arithmetic expression (also called a numeric expression) is a combination of:

- **Numeric Operands**, such as:
  - int & float literal constants (BAD BAD BAD)
  - int & float named constants (GOOD)
  - int & float variables
  - int-valued & float-valued function invocations

- **Arithmetic Operators**, such as:
  - Identity: +
  - Negation: –
  - Addition: +
  - Subtraction: –
  - Multiplication: *
  - Division: /
  - Modulus (remainder): % (only for int operands)

- Parentheses: (  )

---

Some Arithmetic Expression Examples

\[ \begin{align*} 
  &x \\
  &+x \\
  &-x \\
  &x + y \\
  &x - y \\
  &x \times y \\
  &x \div y \\
  &x \% y \\
  &x + y - \frac{z}{22} \times 7 \\
\end{align*} \]
Unary & Binary Arithmetic Operations

Arithmetic operations come in two varieties: unary and binary.

A unary operation is an operation that has only one operand. For example:

\[-x\]

Here, the operand is \(x\), the operator is the minus sign, and the operation is negation.

A binary operation uses two operands. For example:

\[y + z\]

Here, the operands are \(y\) and \(z\), the operator is the plus sign, and the operation is addition.

Arithmetic operations are:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Kind</th>
<th>Operator</th>
<th>Usage</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>Unary</td>
<td>+</td>
<td>+x</td>
<td>Value of (x)</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>-</td>
<td>–x</td>
<td>Additive inverse of (x)</td>
</tr>
<tr>
<td>Negation</td>
<td>Unary</td>
<td>(-)</td>
<td>(x)</td>
<td>Value of (x)</td>
</tr>
<tr>
<td>Addition</td>
<td>Binary</td>
<td>+</td>
<td>(x + y)</td>
<td>Sum of (x) and (y)</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Binary</td>
<td>-</td>
<td>(x - y)</td>
<td>Difference between (x) and (y)</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Binary</td>
<td>(*)</td>
<td>(x \times y)</td>
<td>Product of (x) times (y) (i.e., (x \cdot y))</td>
</tr>
<tr>
<td>Division</td>
<td>Binary</td>
<td>(/)</td>
<td>(x / y)</td>
<td>Quotient of (x) over (y) (i.e., (x \div y))</td>
</tr>
<tr>
<td>Modulus</td>
<td>Binary ((\text{int only}))</td>
<td>(%)</td>
<td>(x % y)</td>
<td>Remainder of (x) over (y) (i.e., (x - \lfloor x / y \rfloor \cdot y))</td>
</tr>
</tbody>
</table>
Structure of Arithmetic Expressions

An arithmetic expression can be long and complicated. For example:

\[ a + b - c \times d \div e \% f \]

Terms and operators can be mixed together in almost limitless variety, but they must follow the rule that a unary operator has a term immediately to its right and a binary operator has terms on both its left and its right:

\[ -a + b - c \times d \div e \% f - 398 + g \times 5981 \div 15 \% h \]

Parentheses can be placed around any unary or binary expression:

\[ ((-a) + b - c) \times d \div e \% ((f-398) + g \times 5981 \div 15) \% h \]

Putting a term in parentheses may change the value of the expression, because a term inside parentheses will be calculated first. For example:

\[ a + b \times c \]

is evaluated as “multiply b by c, then add a”

but

\[ (a + b) \times c \]

is evaluated as “add a and b, then multiply by c”

Note: as a general rule, you cannot put two operators in a row (but we’ll see exceptions, sort of).

**Jargon:**

- **int-valued** & **float-valued** Expressions

An *int-valued expression* is an expression whose result, when evaluated, is an *int*.

A *float-valued expression* is an expression whose result, when evaluated, is a *float*. 
Precedence Order of Arithmetic Operations

In the absence of parentheses that explicitly state the order of operations, the *order of precedence* (also called *order of priority*) is:

- First: multiplication and division, left to right, and then
- Second: addition, subtraction, identity and negation, left to right

After taking into account the above rules, the expression as a whole is evaluated left to right. For example:

- $1 - 2 - 3 \equiv -1 - 3 \equiv -4$ \quad but \quad $1 - (2 - 3) \equiv 1 - (-1) \equiv 2$
- $1 + 2 \times 3 + 4 \equiv 1 + 6 + 4 \equiv 7 + 4 \equiv 11$ \quad but \quad $(1 + 2) \times 3 + 4 \equiv 3 \times 3 + 4 \equiv 9 + 4 \equiv 13$
- $24 \div 2 \times 4 \equiv 12 \times 4 \equiv 48$ \quad but \quad $24 \div (2 \times 4) \equiv 24 \div 8 \equiv 3$
- $5 + 4 \% 6 \div 2 \equiv 5 + 4 \div 2 \equiv 5 + 2 \equiv 7$ \quad but \quad $5 + 4 \% (6 \div 2) \equiv 5 + 4 \% 3 \equiv 5 + 1 \equiv 6$ \quad but \quad $(5 + 4) \% (6 \div 2) \equiv 9 \% (6 \div 2) \equiv 9 \% 3 \equiv 0$

**Rule of Thumb:** if you can’t remember the precedence order of the operators, use lots of parentheses.
Arithmetic Precedence Order Example: int Operands

% cat intexprs.c
#include <stdio.h>

int main ()
{ /* main */
    printf("1 - 2 - 3 = %d\n", 1 - 2 - 3);
    printf("1 - (2 - 3) = %d\n", 1 - (2 - 3));
    printf("\n");
    printf(" 1 + 2 * 3 + 4 = %d\n", 1 + 2 * 3 + 4);
    printf("(1 + 2) * 3 + 4 = %d\n", (1 + 2) * 3 + 4);
    printf("\n");
    printf("24 / 2 * 4 = %d\n", 24 / 2 * 4);
    printf("24 / (2 * 4) = %d\n", 24 / (2 * 4));
    printf("\n");
    printf(" 5 + 4 % 6 / 2 = %d\n", 5 + 4 % 6 / 2);
    printf(" 5 + 4 % (6 / 2) = %d\n", 5 + 4 % (6 / 2));
    printf("(5 + 4) % (6 / 2) = %d\n", (5 + 4) % (6 / 2));
} /* main */
%
% gcc -o intexprs intexprs.c
% intexprs
 1 - 2 - 3 = -4
1 - (2 - 3) = 2
(1 + 2) * 3 + 4 = 13
24 / 2 * 4 = 48
24 / (2 * 4) = 3
5 + 4 % 6 / 2 = 7
5 + 4 % (6 / 2) = 6
(5 + 4) % (6 / 2) = 0

Notice that a printf statement can output the value of an expression.
Arithmetic Precedence Order Example: float Operands

% cat realexprs.c
#include <stdio.h>

int main ()
{ /* main */
  printf("1.0 - 2.0 - 3.0 = %f\n", 1.0 - 2.0 - 3.0);
  printf("1.0 - (2.0 - 3.0) = %f\n", 1.0 - (2.0 - 3.0));
  printf("\n");
  printf(" 1.0 + 2.0 * 3.0 + 4.0 = %f\n",
         1.0 + 2.0 * 3.0 + 4.0);
  printf("(1.0 + 2.0) * 3.0 + 4.0 = %f\n",
         (1.0 + 2.0) * 3.0 + 4.0);
  printf("\n");
  printf("24.0 / 2.0 * 4.0 = %f\n", 24.0 / 2.0 * 4.0);
  printf("24.0 / (2.0 * 4.0) = %f\n", 24.0 / (2.0 * 4.0));
} /* main */
% gcc -o realexprs realexprs.c
% realexprs
 1.0 - 2.0 - 3.0 = -4.000000
 1.0 - (2.0 - 3.0) = 2.000000

  1.0 + 2.0 * 3.0 + 4.0 = 11.000000
 (1.0 + 2.0) * 3.0 + 4.0 = 13.000000

 24.0 / 2.0 * 4.0 = 48.000000
 24.0 / (2.0 * 4.0) = 3.000000

Again, notice that a printf statement can output the value of an expression.
Named Constants & Variables as Operands

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:

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

int 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;
    int year_of_birth, current_year;

    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);
    printf("I’d guess that your age is about\n");
    printf(" %d seconds.\n",
            (current_year - year_of_birth) *
            days_in_a_year * hours_in_a_day *
            minutes_in_an_hour * seconds_in_a_minute);
}
% gcc -o age age.c
% age
Let me guess your age in seconds!
What year were you born?
1946
What year is this?
2003
I’d guess that your age is about 1797552000 seconds.
```

Again, notice that printf statements can output expressions.
Constant-Valued Expressions in Named Constant Initializations

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:

```
% cat constexpr.c
#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 */
%
gcc -o constexpr constexpr.c
%
constexpr
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 constant.
Assignments Using Expressions

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

```
% cat varassn.c
#include <stdio.h>
int main ()
{ /* main */
   int x;
   x = 5;
   printf("x = %d\n", x);
} /* main */
% gcc -o varassn varassn.c
% varassn
x = 5
```

But, the meat and potatoes of computing is the assignment of complicated expressions:

```
% cat triangarea.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 triangarea triangarea.c
% triangarea
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.
```
Using the Same Variable on Both Sides of an Assignment

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

\[ x = y + 1 \]

The compiler interprets the assignment statement to mean, “first, evaluate the expression that’s on the right side of the assignment; then, put the resulting value into the variable that’s on the left side of the assignment.” In this case, the assignment statement means, “evaluate \( y + 1 \), then put the resulting value into \( x \).”

```
% cat xgetsyplus1.c
#include <stdio.h>

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

Here’s another assignment:

\[ x = x + 1; \]

The assignment statement above may be confusing, because it has the same variable, \( x \), on both the left side and the right side of the assignment statement.
The Meaning of Using the Same Variable on Both Sides of an Assignment

\[ x = x + 1; \]

In general, the compiler interprets an assignment statement to mean, “first, evaluate the expression that’s on the right side of the assignment; then, put the resulting value into the variable that’s on the left side of the assignment.”

So, the assignment statement above means, “get the current value of \( x \), then add 1 to it, then place the new value back into \( x \).”

```c
#include <stdio.h>

int 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 assignself assignself.c`
% `assignself`
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 (e.g., all int-valued terms or all float-valued terms), we call this single mode arithmetic.

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

\[
\begin{align*}
5.0 &+ 7.0 \text{ is } 12.0 \text{ and } \\
5 &+ 7 \text{ is } 12 \\
5.0 &- 7.0 \text{ is } -2.0 \text{ and } \\
5 &- 7 \text{ is } -2 \\
5.0 &\times 7.0 \text{ is } 35.0 \text{ and } \\
5 &\times 7 \text{ is } 35
\end{align*}
\]

**But, division is different for ints and floats!**

\[
\begin{align*}
5.0 &/ 7.0 \text{ is } 0.71 \text{ but } \\
5 &/ 7 \text{ is } 0
\end{align*}
\]

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 immediately less than the mathematical result; e.g.,

\[
\begin{align*}
4.0 &/ 4.0 \text{ is } 1.00 \text{ and } \\
4 &/ 4 \text{ is } 1 \\
5.0 &/ 4.0 \text{ is } 1.25 \text{ but } \\
5 &/ 4 \text{ is } 1 \\
6.0 &/ 4.0 \text{ is } 1.5 \text{ but } \\
6 &/ 4 \text{ is } 1 \\
7.0 &/ 4.0 \text{ is } 1.75 \text{ but } \\
7 &/ 4 \text{ is } 1 \\
8.0 &/ 4.0 \text{ is } 2.0 \text{ and } \\
8 &/ 4 \text{ is } 2
\end{align*}
\]
Division By Zero

Mathematically, division by zero gives an infinite result:

\[
\frac{c}{0} = \infty \text{ for } c \neq 0
\]

Computationally, division by zero causes an error.

```c
#include <stdio.h>

int main ()
{
    /* main */
    printf("5 / 0 = %d\n", 5 / 0);
} /* main */
```

If you manage to slip a zero denominator past the compiler, the program will crash:

```c
#include <stdio.h>

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

    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);
    printf("numerator / denominator = %d\n", numerator / denominator);
} /* main */
```

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

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

In practice, though, we can freely mix int-valued and float-valued literals, variables and expressions, subject to the rule that an operation with operands of both types has a float result.

We call such expressions mixed mode arithmetic.

\[
\begin{align*}
2 + 1 & \text{ is } 3 \quad \text{and} \\
2 + 1.0 & \text{ is } 3.0 \\
2 - 1 & \text{ is } 1 \quad \text{and} \\
2 - 1.0 & \text{ is } 1.0 \\
2 * 1 & \text{ is } 2 \quad \text{and} \\
2 * 1.0 & \text{ is } 2.0 \\
1 / 2 & \text{ is } 0 \quad \text{but} \\
1 / 2.0 & \text{ is } 0.5 \\
\end{align*}
\]

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

\[
\begin{align*}
1 / 2 & \text{ is } 0 \quad \text{but} \\
1 / 2.0 & \text{ is } \\
1.0 / 2.0 & \text{ is } 0.5 \\
4.0 / (3 / 2) & \text{ is } 4.0 \quad \text{but} \\
4.0 / (3 / 2.0) & \text{ is } \\
4.0 / (3.0 / 2.0) & \text{ is } 2.666... \\
\end{align*}
\]
Exercise: Writing a Program

Let $c$ be the temperature in degrees Celsius. Let $f$ be the temperature in degrees Fahrenheit. Then:

$$f = \frac{9}{5} c + 32$$

Write a C program that prompts the user and then inputs a temperature in degrees Celsius, calculates the temperature in degrees Fahrenheit, then outputs the temperature in degrees Fahrenheit.

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.