### Arithmetic Expressions Lesson #1 Outline

1. Arithmetic Expressions Lesson #1 Outline
2. A Less Simple C Program #1
3. A Less Simple C Program #2
4. A Less Simple C Program #3
5. A Less Simple C Program #4
6. A Less Simple C Program: Compile & Run
7. Flowchart for my_add.c
8. Named Constant Example Program
9. Named Constant Example Program
10. 1997 Tax Program with Named Constants
11. What is an Expression? #1
12. What is an Expression? #2
13. What is an Expression? #3
14. What is an Expression? #4
15. What is an Expression? #5
16. What is an Expression? #6
17. What is an Arithmetic Expression? #1
18. What is an Arithmetic Expression? #2
19. What is an Arithmetic Expression? #3
20. Arithmetic Expression Examples
22. Arithmetic Operations
23. Structure of Arithmetic Expressions #1
24. Structure of Arithmetic Expressions #2
25. int-valued & float-valued Expressions
26. Precedence Order
27. Precedence Order Examples
28. Precedence Order Example: int #1
29. Precedence Order Example: int #2
30. Precedence Order Example: float #1
31. Precedence Order Example: float #2
A Less Simple C Program #1

/*
 *****************************************
 *** Program: my_add ***
 *** Author: Henry Neeman (hneeman@ou.edu) ***
 *** Course: CS 1313 010 Fall 2019 ***
 *** Lab: Sec 014 Fridays 2:30pm ***
 *** Description: Input two integers, compute ***
 *** their sum and output the result. ***
 *****************************************
*/
#include <stdio.h>
int main ()
{ /* main */
  /*
  *****************************************
  *** Declaration Section ***
  *****************************************

  */
  const int program_success_code = 0;

  /*
  *****************************************
  *** Declaration Section ***
  *****************************************

  */
  int addend, augend, sum;

  Continue on the next slide.
A Less Simple C Program #2

/*
 * ***********************************
 * *** Execution Section ***
 * ***********************************
 * 
 * ***********************************
 * Greeting Subsection *
 * ***********************************
 * 
 * Tell the user what the program does.
 * */
 printf("I'll add a pair of integers.\n");
/*
 * ***********************************
 * Input subsection *
 * ***********************************
 * 
 * Prompt the user to input the addend & augend.
 * */
 printf("What pair of integers do you want to add?\n");
/*
 * Input the integers to be added.
 * */
 scanf("%d %d", &addend, &augend);

Continued on the next slide.
The statement as a whole is an assignment statement.
The stuff to the right of the single equals sign is an arithmetic expression.
#include <stdio.h>

int main ()
{ /* main */
    const int program_success_code = 0;
    int addend, augend, sum;

    printf("I'll add a pair of integers.\n");
    printf("What pair of integers do you want to add?\n");
    scanf("%d %d", &addend, &augend);
    sum = addend + augend;
    printf("The sum of %d and %d is %d.\n",
            addend, augend, sum);
    return program_success_code;
} /* main */

The statement as a whole is an **assignment statement**.

The stuff to the right of the single equals sign is an **arithmetic expression**.
A Less Simple C Program: Compile & Run

% gcc -o my_add my_add.c
% my_add
I'll add a pair of integers.
What pair of integers do you want to add?
5 7
The sum of 5 and 7 is 12.
% my_add
I'll add a pair of integers.
What two integers do you want to add?
1593
09832
The sum of 1593 and 9832 is 11425.
Flowchart for `my_add.c`

A rectangle denotes an operation other than I/O or branching (for example, calculation).
Named Constant Example Program

```
% cat circlecalc.c
#include <stdio.h>
int main ()
{ /* main */
    const float pi                   = 3.1415926;
    const float diameter_factor     = 2.0;
    const int    program_success_code = 0;
    float radius, circumference, area;

    printf("I'm going to calculate a circle's\n");
    printf(" circumference and area.\n");
    printf("What's the radius of the circle?\n");
    scanf("%f", &radius);
    circumference = pi * radius * diameter_factor;
    area = pi * radius * radius;
    printf("The circumference is %f\n", circumference);
    printf(" and the area is %f.\n", area);
    return program_success_code;
} /* main */
% gcc -o circlecalc circlecalc.c
% circlecalc
I'm going to calculate a circle's circumference and area.
What's the radius of the circle?
5
The circumference is 31.415924 and the area is 78.539810.
```
Named Constant Example Program

```c
#include <stdio.h>

int main ()
{ /* main */
    const float pi                   = 3.1415926;
    const float diameter_factor = 2.0;
    const int      program_success_code = 0;
    float radius, circumference, area;

    printf("I'm going to calculate a circle's\n");
    printf(" circumference and area.\n");
    printf("What's the radius of the circle?\n");
    scanf("%f", &radius);
    circumference = pi * radius * diameter_factor;
    area = pi * radius * radius;
    printf("The circumference is %f\n", circumference);
    printf(" and the area is %f.\n", area);
    return program_success_code;
} /* main */
```

```
gcc -o circlecalc circlecalc.c

circlecalc

I'm going to calculate a circle's circumference and area.
What's the radius of the circle?
5
The circumference is 31.415924 and the area is 78.539810.
```
1997 Tax Program with Named Constants

```c
#include <stdio.h>

int main ()
{
  const float standard_deduction = 4150.0;
  const float single_exemption = 2650.0;
  const float tax_rate = 0.15;
  const int tax_year = 1997;
  const int program_success_code = 0;
  float income, tax;

  printf("I'm going to calculate the federal income tax\n");
  printf(" on your %d income.\n", tax_year);
  printf("What was your %d income in dollars?\n", tax_year);
  scanf("%f", &income);

  tax = (income - (standard_deduction + single_exemption)) * tax_rate;

  printf("The %d federal income tax on $%2.2f\n", tax_year, income);
  printf(" was $%2.2f.\n", tax);
  return program_success_code;
}
```

```bash
% gcc -o tax1997_named tax1997_named.c
% tax1997_named
I'm going to calculate the federal income tax
on your 1997 income.
What was your 1997 income in dollars?
20000
The 1997 federal income tax on $20000.00
  was $1980.00.
```
What is an Expression? #1

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

- **Operands**
- **Operators**
- **Parentheses**: (   )

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

**NOTE:** In C, the only characters you can use for parenthesizing are **actual parentheses** (unlike in math, where you can also use square brackets and curly braces as well.)
What is an Expression? #2

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

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

- **Operators**

- **Parentheses**: (  )
What is an Expression? #3

In programming, an expression is a combination of:

- **Operands**
- **Operators**, such as:
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
- **Parentheses**: (  )
What is an Expression? #4

In programming, an expression is a combination of:

- **Operands**
- **Operators**, such as:
  - Arithmetic Operators
    - Addition: \(+\)
    - Subtraction: \(-\)
    - Multiplication: \(*\)
    - Division: \(/\)
    - **Modulus** (remainder): \(\%\) (only for int operands)
  - Relational Operators
  - Logical Operators
- **Parentheses**: \((\ )\)
What is an Expression? #5

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

- **Operands**
- **Operators**, such as:
  - Arithmetic Operators
  - Relational Operators
    - Equal: `==`
    - Not Equal: `!=`
    - Less Than: `<`
    - Less Than or Equal To: `<=`
    - Greater Than: `>`
    - Greater Than or Equal To: `>=`
  - Logical Operators
- **Parentheses**: `()`
What is an Expression? #6

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

- **Operands**
- **Operators**, such as:
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
    - *Negation* (NOT): !
    - *Conjunction* (AND): &&
    - *Disjunction* (OR): ||
- **Parentheses**: (  )

We’ll learn about these later.
What is an Arithmetic Expression? #1

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

- *Numeric operands*
- *Arithmetic Operators*
- *Parentheses*: (  )
What is an Arithmetic Expression? #2

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**

- **Parentheses**: ( )
What is an Arithmetic Expression? #3

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

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

\[
x + x
\]
\[
-x
\]
\[
x + y
\]
\[
x - y
\]
\[
x * y
\]
\[
x / y
\]
\[
x % y
\]
\[
x + y - (z \% 22) * 7 / \cos(\theta)
\]
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

<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>Negation</td>
<td>Unary</td>
<td>-</td>
<td>-x</td>
<td>Additive inverse 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 * y</td>
<td>Product of x times y (i.e., x · y)</td>
</tr>
<tr>
<td>Division</td>
<td>Binary</td>
<td>/</td>
<td>x / y</td>
<td>Quotient of x divided by y (i.e., x ÷ y)</td>
</tr>
<tr>
<td>Modulus</td>
<td>Binary</td>
<td>%</td>
<td>x % y</td>
<td>Remainder of x divided by y (i.e., x - x ⌊x ÷ y⌋ · y)</td>
</tr>
</tbody>
</table>
Structure of Arithmetic Expressions #1

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 subexpression:

\[ (\neg a) + b - c \times d \div e \% f - (398 + g) \times 5981 \div 15 \% h \]
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).
int-valued & float-valued Expressions

An **int-valued expression** is an expression that, when it is evaluated, has an **int** result.

A **float-valued expression** is an expression that, when it is evaluated, has a **float** result.
Precedence Order

In the absence of parentheses that explicitly state the order of operations, the *order of precedence* (also known as the *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.
Precedence Order Examples

- $1 - 2 - 3 = -1 - 3 = -4$ but $1 - (2 - 3) = 1 - (-1) = 2$
- $1 + 2 * 3 + 4 = 1 + 6 + 4 = 7 + 4 = 11$ but $(1 + 2) * 3 + 4 = 3 * 3 + 4 = 9 + 4 = 13$
- $24 / 2 * 4 = 12 * 4 = 48$ but $24 / (2 * 4) = 24 / 8 = 3$
- $5 + 4 \% 6 / 2 = 5 + 4 / 2 = 5 + 2 = 7$ but $5 + 4 \% (6 / 2) = 5 + 4 \% 3 = 5 + 1 = 6$ but $(5 + 4) \% (6 / 2) = 9 \% (6 / 2) = 9 \% 3 = 0$

**Rule of Thumb:** If you can’t remember the precedence order of the operations, use lots of parentheses.

But **DON’T** overdo your use of parentheses, because then your code would be “write only.”
Precedence Order Example: int #1

#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 */

Notice that a printf statement can output the value of an expression.
Precedence Order Example: int #2

```plaintext
% gcc -o int_expressions int_expressions.c
% int_expressions
1 - 2 - 3 = -4
1 - (2 - 3) = 2

1 + 2 * 3 + 4 = 11
(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
```
Precedence Order Example: float #1

```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 */
```

Again, notice that a `printf` statement can output the value of an expression.
Precedence Order Example: float #2

% gcc -o real_expressions real_expressions.c
% real_expressions

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