Expressions and Operators in C

Expressions are Used to Perform Calculations

Let's talk in more detail starting with a fifth element of C syntax: expressions. An expression is a calculation consisting of variables and operators.* For example,

\[
\begin{align*}
A + 42 \\
A / B
\end{align*}
\]

Deposits – Withdrawals

* And function calls, but that topic we leave for ECE220.

Our Class Focuses on Four Types of Operator in C

The C language supports many operators. In our class, we consider four types:

* arithmetic operators
* bitwise Boolean operators
* relational/comparison operators
* the assignment operator

We also introduce logical operators, but leave their full meaning for ECE220.

Five Arithmetic Operators on Numeric Types

Arithmetic operators in C include

* addition: +
* subtraction: –
* multiplication: *
* division: /
* modulus: % (integers only)

The C library includes many other functions, such as exponentiation, logarithms, square roots, and so forth. We leave these for ECE220.
**Arithmetic Mostly Does What You Expect**

Declare: \( \text{int } A = 120; \text{ int } B = 42; \)

Then...

- \( A + B \) evaluates to 162
- \( A - B \) evaluates to 78
- \( A \times B \) evaluates to 5040
- \( A \% B \) evaluates to 36
- \( A / B \) evaluates to... 2

What's going on with division?

---

**A Few Pitfalls of C Arithmetic**

- **No checks for overflow**, so be careful.
  - unsigned \( \text{int } A = 0 - 1; \)
  - \( A \) is a large number!

- **Integer division**
  - Trying to **divide by 0** ends the program (floating-point produces **infinity** or **NaN**).
  - Integer division evaluates to an integer, so 
    \( (100 / 8) \times 8 \) is **not** 100.

---

**C Behavior Sometimes Depends on the Processor**

- Integer division is rounded to an integer.
- Rounding depends on the processor.
- Most modern processors round towards 0, so...
  - \( -11 / 3 \) evaluates to 3
  - \( -11 / 3 \) evaluates to -3
- Modulus \( A \% B \) is defined such that
  \( (A / B) \times B + (A \% B) \) is equal to \( A \)
- So \( -11 \% 3 \) evaluates to -2.
  
  **Modulus is not always positive.**

---

**Six Bitwise Operators on Integer Types**

- Bitwise operators in C include
  - **AND:** \( \& \)
  - **OR:** \( | \)
  - **NOT:** \( ~ \)
  - **XOR:** \( ^ \)
  - **left shift:** \( < < \)
  - **right shift:** \( >> \)

- In some languages, \(^\) means exponentiation, but not in the C language.
Bitwise Operators Treat Numbers as Bits

Declare: int A = 120; int B = 42;
/* A = 0x00000078, B = 0x0000002A
using C’s notation for hexadecimal. */

Then...
A & B evaluates to 40 0x00000028
Apply AND to pairs of bits.

Left Shift by N Multiplies by $2^N$

Shifting left by N bits adds N 0s on right.
* It’s like multiplying by $2^N$.
* N bits lost on left! (Shifts can overflow.)

Declare: int A = 120; /* 0x00000078 */
unsigned int B = 0xFFFFFFFF;

Then...
A << 2 evaluates to 480 0x000001E0
B << 4 evaluates to (<B!) 0xFFFF0000

Right Shift by N Divides by $2^N$

A question for you: What bits appear on the left when shifting right?

Declare: int A = 120; /* 0x00000078 */

A >> 2 evaluates to 30 0x000001E

What about 0xFFFFFFFF >> 4?

Is 0xFFFFFFFF equal to
-256 (/16 = -16, so insert 1s)? or equal to
4,294,967,040 (/16 = 268,435,440, insert 0s)?
Right Shifts Depend on the Data Type

A C compiler uses the type of the variable to decide which type of right shift to produce.

For an int:
- 2’s complement representation
- produces arithmetic right shift
- (copies the sign bit)

For an unsigned int:
- unsigned representation
- produces logical right shift
- (inserts 0s on left)

Right Shift by N Divides by $2^N$

Declare: `int A = -120; /* 0xFFFFFF88 */
unsigned int B = 0xFFFFFF00;

Then...
- A >> 2 evaluates to -30 0xFFFFFE2
- A >> 10 evaluates to -1 0xFFFFFFFF
- B >> 2 evaluates to 0x3FFFFFC0
- B >> 10 evaluates to 0x003FFFFFF

Notice that right shifts round down.

Six Relational Operators

Relational operators in C include:
- less than: `<`
- less or equal to: `<=`
- equal: `==` (TWO equal signs)
- not equal: `!=`
- greater or equal to: `>`
- greater than:

C operators cannot include spaces, nor can they be reordered (so no "< =" nor "=<").

Relational Operators Evaluate to 0 or 1

In C,  
- 0 is false, and  
- all other values are true.

Relational operators always  
- evaluate to 0 when false, and  
- evaluate to 1 when true.
Relational Operators Also Depend on Data Type

Declare:

```c
int A = -120; /* 0xFFFFFF88 */
int B = 256; /* 0x00000100 */
```

Is A < B?
- Yes, -120 < 256.
- But if the same bit patterns were interpreted using the unsigned representation,
  ```c
  0xFFFFFF88 > 0x00000100
  ```
As with shifts, a C compiler uses the data type to perform the correct comparison.

The Assignment Operator Can Change a Variable's Value

The C language uses `=` as the assignment operator. For example,

```c
A = 42
```
changes the bits of variable A to represent the number 42.

One can write any expression on the right-hand side of assignment. So

```c
A = A + 1
```
increments the value of variable A by 1.

Only Assign Values to Variables

A C compiler can not solve equations.
For example,

```c
A + B = 42
```
results in a compilation error (the compiler cannot produce instructions for you).

The left-hand side of an assignment must be a variable.*

* For ECE120, ECE220 teaches other ways to use the assignment operator.

Pitfall of the Assignment Operator

Programmers sometimes
- write “=” (assignment)
- instead of “==” (comparison for equality).

For example, to compare variable A to 42,
- one might want to write “A == 42”
- but instead write “A = 42” by accident.
A C compiler can sometimes warn you (in which case, fix the mistake!).
Good Programming Habits Reduce Bugs

To avoid these mistakes, get in the habit of writing comparisons with the variable on the right.
For example, instead of “A == 42”, write
42 == A
If you make a mistake and write “42 = A”,
• the compiler will always tell you,
• and you can fix the mistake.

Three Logical Operators

Logical operators in C include
• AND: &&
• OR: ||
• NOT: !
Logical operators operate on truth values (again, 0 is false, and non-zero is true).
Logical operators
• evaluate to 0 (false), or
• evaluate to 1 (true).

Logical Operators Depend only on True/False in Operands

Declare: int A = 120;  int B = 42;
Then...
(0 > A || 100 < A) evaluates to 1
(120 == A && 3 == B) evaluates to 0
!(A == B) evaluates to 1
!(0 < A && 0 < B) evaluates to 0
(B + 78 == A) evaluates to 1
(So no bitwise calculations, just true/false.)

Operator Precedence in C is Sometimes Obvious

A task for you:
Evaluate the C expression: 1 + 2 * 3
Did you get 7?
Why not 9? (1 + 2) * 3
Multiplication comes before addition
• in elementary school
• and in C!
The order of operations is called operator precedence.
Never Look Up Precedence Rules!

Another task for you:

**Evaluate the C expression:** 10 / 2 / 3

Did you get 1.67?
Is it a friend's birthday?
Perhaps it causes a divide-by-0 error?
Or maybe it's ... 1? (10 / 2) / 3, as int

**If the order is not obvious,**
- Do NOT look it up.
- Add parentheses!