University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Expressions and Operators in C
(Partially a Review)

Expressions are Used to Perform Calculations

An **expression** is a calculation consisting of variables, operators, and function calls.

For example,

A + 42

A / B

Deposits – Withdrawals

scanf ("%f", &flt)

Our Class Focuses on Four Types of Operator in C

The **C** language supports many operators.

In ECE120, you learned about four types:
- **arithmetic** operators
- **bitwise** Boolean operators
- **relational** / **comparison** operators
- the **assignment** operator

Let's review those first.

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.

Arithmetic Mostly Does What You Expect

Declare: `int A = 120;  int B = 42;`

Then…

`A + B`	**evaluates to**	**162**
`A – B`	evaluates to	**78**
`A * 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 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) * 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) * 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 exponentation, 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`**

```
    0000 0000 0000 0000 0000 0000 0111 1000
AND 0000 0000 0000 0000 0000 0000 0010 1010
    ---------------------------------------
    0000 0000 0000 0000 0000 0000 0010 1000
```

Apply AND to pairs of bits.

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`**
`A | B`	evaluates to	**122**	**`0x0000007A`**
`~A`	evaluates to	**-121**	**`0xFFFFFF87`**
`A ^ B`	evaluates to	**82**	**`0x00000052`**

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 = 0xFFFFFF00;`

Then…

`A << 2`	evaluates to	**480**	**`0x000001E0`**
`B << 4`	evaluates to	**(<B!)**	**`0xFFFFF000`**

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** **`0x0000001E`**

What about **`0xFFFFFF00 >> 4`**?

Is **`0xFFFFFF00`** 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**	**0xFFFFFFE2**
`A >> 10`	evaluates to	**-1**	**0xFFFFFFFF**
`B >> 2`	evaluates to		**0x3FFFFFC0**
`B >> 10`	evaluates to		**0x003FFFFF**

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:
```
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,

  `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,

`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

`A = A + 1`

increments the value of variable **A** by **1**.

Assignment Calculates an Expression, then Writes Bits

The code for an assignment

1. **calculates the expression**, then
2. **writes the result to the address** for the left-hand side.

For example, given

`A = B + C`

A compiler produces something akin to this code.

```
LD R0,B
LD R1,C
ADD R0,R0,R1
ST R0,A
```


Assignments Write to Memory Addresses

A **C** compiler can not solve equations.

For example,

`A + B = 42`

results in a compilation error (the compiler cannot produce instructions for you).

The left-hand side of an assignment must have an address.

An expression with an address is called an **l-value**. **Variables are l-values.**

Assignments Evaluate to their Right-Hand Side

Note: **an assignment is an expression**.

Assignment **evaluates to** the value of the **right-hand side**.

So, for example, one can write:

```
A = B = 0; // same as A = (B = 0);
```

The expression "`B = 0`" evaluates to 0, so A is also assigned the value 0.

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.

Operator Precedence in C is Sometimes Obvious

A task for you:

Evaluate the C expression: ~~`1 + 2 * 3`~~
`10 + 4 * 8`

Did you get 42?

Why not 112? (10 + 4) × 8

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**!

Compiler Silently Auto-Converts … Sometimes

What does this code do?

```
int x;

x = 3 + 4.6;
```

constant has type **`int`**

constant has type **`double`**

`x` has type **`int`**

1. Convert 3 to **`double`**.
2. Add two **`double`**s.
3. Convert sum to **`int`** (truncates to 7).
4. Stores 7 in x.

Be Careful with Auto-Conversion

How does auto-conversion work?

When there's a choice, into the "larger" type.

What does that mean? Nothing obvious.

Integers convert to floating-point.

```
unsigned a = 10;
int b = -20;
if (a + b < 0) {
    printf ("ok");
}
```

What does the code to the left print?

Nothing.
As you'd expect?

Be Careful with Auto-Conversion

Auto-conversion happens silently:
no errors, and no warnings.

For anything unclear (anything with a choice), avoid auto-conversion, or use explicit conversions (example to right).

```
unsigned a = 10;
int b = -20;
if (((int)a) + b < 0) {
    printf ("ok");
}
```


Now Consider Three New Kinds of Operators

Let's consider some new operators
(we'll learn more later, too).

Let's look at these:
- logical operators (and shortcutting)
- conditional operator
- modification operators

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.)

Remember these Simple Boolean Properties?

Easy, but useful to commit to memory for analyzing circuits…

$1 + A = 1$	$0 \cdot A = 0$
$1 \cdot A = A$	$0 + A = A$
$A + A = A$	$A \cdot A = A$
$A \cdot A' = 0$	

(Each row give

Remember these Boolean properties from ECE120?

Logical Operators Shortcut Evaluation in C

In **C**,
- **logical AND and OR**
- **stop evaluating operands**
- **when the operator's result is known**.

For example,

```
0 && this_function_crashes ()
```

does NOT call the function.

The **first operand is false** (0 in **C**),
so the **second operand** (the function call) is
not evaluated.

Logical AND Stops on False, Logical OR Stops on True

Similarly, if we write

```
1 || this_function_crashes ()
```

does NOT call the function.

The **first operand is true** (not 0 in **C**),
so the **second operand** (the function call) is
not evaluated.

Use Shortcutting to Protect Unsafe/Undesired Actions

Here's a more realistic example…

```
if (1 == scanf ("%d", &age) &&
    0 <= printf ("Salary? ") &&
    1 == scanf ("%d", &salary)) {
    // use age and salary
}
```

scanf in these cases returns 1 on success,
and **printf** returns 8 (characters) on success.

Use Shortcutting to Protect Unsafe/Undesired Actions

And another one…

```
if (0 <= dist_sq &&
    walk_p (me, sqrt (dist_sq))) {
    // go for a walk
}
```

Calculating the square root (**sqrt**) of a
negative number may cause a crash.

Conditional Operator is Shorthand for If/Then/Else

The code to the right
- **assigns one of two values** to **A**
- **based on a condition**.

```
if (B > 0) {
    A = C;
} else {
    A = D;
}
```

C provides a conditional operator for this type of construct:

```
A = (B > 0 ? C : D);
```


Increment and Decrement Change Integer Variables

C provides two operators to
- **increment (++)** and
- **decrement (--)**
- integer variables.

One can write either operator before (pre-) or after (post-) a variable name.

```
int i;
i++; // Used by themselves,
++i; // these are identical.
```


Read Increment and Decrement from Left to Right

The **difference** in pre- and post- versions **arises when one uses the value of the expression**.

Read left to right:
- **i++ : read the value, then increment i**
- **++i: increment i, then read it**

Example of Pre- and Post-Increment

For example …

```
int i = 18;
int j = 23;
int k;

k = (++i) + (j++);
```

What are i, j, and k afterward?

i is 19, j is 24, and k is 42 (19 + 23).

Example of Pre- and Post-Increment

How about this one?

```
int i = 18;
int j = 23;

j = (++i) + (j++)
    + (++j)
    - (i--);
```

What are i, and j afterward?

Who cares?* Don't write such code!

*The result is perhaps even undefined, meaning that different compilers can generate different results.

Modification Operators: Shorthand for Binary Operators

C supports

- many modification operators for variables.
- These are simply shorthand.

For example,

```
A += B;    // same as A = A + B
A &= MASK; // same as A = A & MASK
```

(others: `-=`, `*=`, `/=`, `%=`, `|=`, `^=`, `<<=`, `>>=`)

