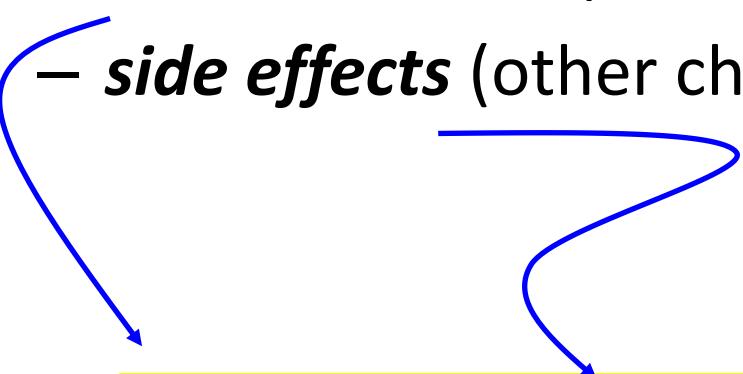


# C Expressions and Operators

ITSC 2181: Introduction to Computer Systems  
UNC Charlotte  
College of Computing and Informatics

# Expressions

- Most statements in a C program are *expressions*
- ***Evaluating*** an expression means doing the computation according to the definition of the operations specified
- **Results** of expression evaluation:
  - **value** returned (and assigned); **and/or**
  - ***side effects*** (other changes to variables, or output, along the way)



```
j = k + 3 * m++;
```

# What Are the C Operators?

- There are approximately 50 of them
- Categories of operators
  1. arithmetic
  2. logical and relational
  3. assignment
  4. bitwise operators
  5. “other”

# Arithmetic: Single Operand Operators

Unary plus (**+a**): no effect

```
a = +b;
```

(see **expressions.c** in *Code samples*  
and *Demonstrations* in *Canvas*)

Unary minus (**-b**): changes sign of operand

```
a = -b;
```

Increment (**++**) and decrement (**--**) operators

- operand type must be modifiable (not a constant)
- these operators have side effects!

```
a = ++b / c-- ;
```

# Single Operand... (cont'd)

**prefix:** side effect takes place first, then expression value is determined

```
int i = 1, j = 8;  
printf("%d %d\n", ++i, --j);  
printf("%d %d\n", i, j);
```

what is the output?

**postfix:** expression uses old operand value first, then side effect takes place

```
int i = 1, j = 8;  
printf("%d %d\n", i++, j--);  
printf("%d %d\n", i, j);
```

what is the output?

(see [expressions.c](#) in Code samples  
and Demonstrations in Canvas)

common source of bugs  
**difference between  
postfix and prefix**



COLLEGE OF COMPUTING  
AND INFORMATICS

# Arithmetic on Two Operands

- Multiplication (\*), Quotient (/), Remainder (%), Addition (+), Subtraction (-)
  - Possibility of underflow and overflow during expression evaluation, or assignment of the results
- Division by zero
  - causes program execution failure if the operands are of integer type
  - generates a special value (**inf**) and continues execution if the operands are IEEE floating point

✖ common source of bugs ✖  
**overflow in computations**

(see **expressions.c** in  
Code samples and  
Demonstrations in Canvas)

✖ common source of bugs ✖  
**divide by zero**

# Arithmetic on Two Operands

- Modulus operator (%) operands **must** have type integer, **should** both be positive

```
printf("%d", (37 % 3));
```

results?

```
printf("%d", (-37 % 3));
```

- Result of **a % b** is a program exception if **b == 0**

(see [expressions.c](#) in Code samples  
and Demonstrations in Canvas)

# Assignment Operators

- **a = b** assigns the value of **b** to **a**
  - **a** must be a reference and must be *modifiable* (**not** a function, **not** an entire array, etc.)
- Both **a** and **b** must be one of the following
  - **numbers** (integer or floating), or
  - **structs** or **unions** of the same type, or
  - **pointers** to variables of the same type

**OK**

```
float a;  
int b = 25;  
a = b;
```

**Not OK**

```
float a[2];  
int b[2] = {25, 15};  
a = b;
```

# Assignment Operators (cont'd)

- **a op= b**
  - where **op** is one of `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|`
  - “shorthand” for **a = a op b**

```
int i = 30, j = 40, k = 50;  
i += j; // same as i = i + j  
k %= j; // same as k = k % j  
j *= k; // same as j = j * k
```

# Constant Expressions

- Constant-valued expressions are used in...

- case statement labels
- array bounds
- bit-field lengths
- values of enumeration constants
- initializers of **static** variables

all evaluated at  
**compile** time,  
not run time

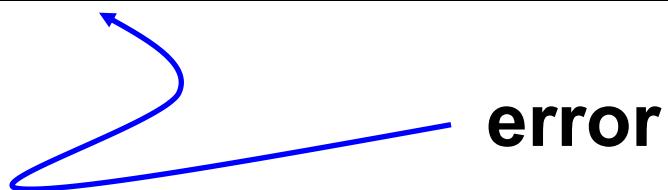
```
static int a = 35 + (16 % (4 | 1));
```

*(static: variable's value is initialized only once, no matter how many times the block in which it is defined is executed)*

# Constant Expressions... (cont'd)

- **Cannot** contain assignments, increment or decrement operators, function calls, ...
  - see a C reference manual for all the restrictions
  - basically: nothing that has to be evaluated at **run-time**

```
static int b = a++ - sum();
```



error

# C Operator Precedence

Tokens	Operator	Class	Prec.	Associates
<b>a[k]</b>	subscripting	postfix	16	left-to-right
<b>f( . . . )</b>	function call	postfix		left-to-right
.	direct selection	postfix		left-to-right
->	indirect selection	postfix		left to right
++ --	increment, decrement	postfix		left-to-right
++ --	increment, decrement	prefix		right-to-left
<b>sizeof</b>	size	unary	15	right-to-left
~	bit-wise complement	unary		right-to-left
!	logical NOT	unary		right-to-left
- +	negation, plus	unary		right-to-left
&	address of	unary		right-to-left
*	Indirection ( <i>dereference</i> )	unary		right-to-left

## C Operator Precedence (cont'd)

<b>(type)</b>	casts	unary	<b>14</b>	right-to-left
* / %	multiplicative	binary	<b>13</b>	left-to-right
+ -	additive	binary	<b>12</b>	left-to-right
<< >>	left, right shift	binary	<b>11</b>	left-to-right
< <= > >=	relational	binary	<b>10</b>	left-to-right
== !=	equality/ineq.	binary	<b>9</b>	left-to-right
&	bitwise and	binary	<b>8</b>	left-to-right
^	bitwise xor	binary	<b>7</b>	left-to-right
	bitwise or	binary	<b>6</b>	left-to-right
&&	logical AND	binary	<b>5</b>	left-to-right
	logical OR	binary	<b>4</b>	left-to-right
? :	conditional	ternary	<b>3</b>	right-to-left
= += -= *= /= %= &= ^=  = <<= >>=	assignment	binary	<b>2</b>	right-to-left
,	sequential eval.	binary	<b>1</b>	left-to-right

# Order of Evaluation in Compound Expressions

- Which operator has higher **precedence**?
- If two operators have equal precedence, are operations evaluated **left-to-right** or **right-to-left**?
- Example:

```
a += b = q - ++ r / s && ! t == u ;
```

what gets executed first, second, ...?

One solution: use parentheses to force a specific order

```
t = (u + v) * w;
```

# Order of Evaluation in Compound Expressions

- **Common mistake:** overlooking precedence and associativity (l-to-r or r-to-l)

```
t = u+v * w;
```

✖ common source of bugs ✖  
**failure to use parentheses  
to enforce precedence**

Advice: either...

- force order of evaluation when in doubt by **using parentheses**
- or (even better) write one large expression as sequence of **several smaller expressions**

# Evaluating Expressions... (cont'd)

- Instead of...

```
a+=b=q-++r/ (s^!t==u) ;
```

*common source of bugs*  
**expressions that  
are too complex**

Or...

```
a+=(b=(q-((++r)/(s^((!t)==u))))));
```

Better:

```
tmp1 = s ^ ( (!t) == u );
tmp2 = (++r) / tmp1;
b = q - tmp2;
a += b;
```

# The C Conditional Operator

- A terse way to write if-then-else statements

```
c = (a > b) ? d : e;
```

- This is equivalent to (shorthand for)

```
if (a > b)
    c = d;
else
    c = e;
```

*\* common source of bugs \**  
**complex conditional statements**

# References

- S. J. Matthews, T. Newhall and K. C. Webb, *Dive into Systems*, Version 1.2. Free online textbook, available at:  
<https://diveintosystems.org/book/>
- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.
- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.