C Expressions, Operators, and Flow of Control

C Programming and Software Tools

N.C. State Department of Computer Science

Computer Science

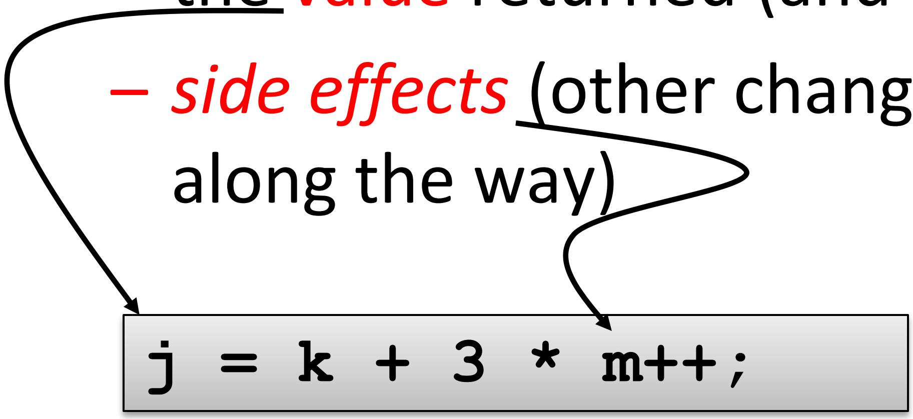
NC STATE UNIVERSITY

Outline

- Expressions
- Operators
  - Single operand
  - Two operands
  - Relational
  - Logical
  - Assignment
- Statement Separation
- C Operator Precedence and Order of Evaluation
- Flow of Control

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
  - the value returned (and assigned); and/or
  - *side effects* (other changes to variables, or output, along the way)



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

Comparison: C vs. Java Operators

Operator	Description	Associates
.	access class feature	left-to-right
a[]	array index	
fn()	function call	
++ --	post-inc/dec	
++ --	pre-inc/dec	right-to-left
~	bitwise not	
!	logical not	
- +	unary -/+	
& *	address/dereference	
(type)	cast	
~~new~~	object allocation	

Not in C

Operator	Description	Associates
`* / %`	multiplicative	left-to-right
`+ -`	additive	left-to-right
`<< >>` ~~`>>>`~~	left, right shift	left-to-right
`< <= > >=`	relational	left-to-right
`== !=`	equality/ineq.	left-to-right
~~`instanceof`~~	test object type	
`&`	bitwise and	left-to-right
`^`	bitwise xor	left-to-right
`|`	bitwise or	left-to-right
`&&`	logical and	left-to-right
`||`	logical or	left-to-right

`= += -=`
`*= /= %=`
`&= ^= |=`
`<<= >>=`
	assignment	right-to-left

- C operators not found in Java:
  - pointer operations (`->`, `&`, `*`)
  - sizeof
  - sequential evaluation (,)

What Are the C Operators?

- There are approximately <u>50</u> of them
- Most operators do the same thing in Java and C
- Categories of operators
  1. “other”
  2. arithmetic
  3. logical and relational
  4. assignment
  5. bit operators

Other Operators

- Array indexing (`x[]`) Later
- Function calls (`f()`) Covered
- Address-of (`&x`) operator, and pointer dereferencing (`*x`) Later
  - and effect of other operators on pointers
- Member (of `struct`) specification Later
  - direct (`x.y`) and indirect (`x->y`)
- The `sizeof()` operator Covered
- casting: `(type) operand` Covered

Arithmetic: Ops on a Single Operand

Unary plus (+a): no effect

```
a = +b;
```

Unary minus (-b) : change 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 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 output?

☠ *common source of bugs* ☠

difference between postfix and prefix

Arithmetic on Two Operands

- Multiply (*), Quotient (/), Remainder (%), Add (+), Subtract (-)
  - possibility of underflow and overflow during expression evaluation, or assignment of the results

☠ *common source of bugs* ☠

overflow in computations

- Divide by zero
  - causes program execution failure if the operands are integer type
  - generates a special value (inf) and continues execution if the operands are IEEE floating point

☠ *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));
```

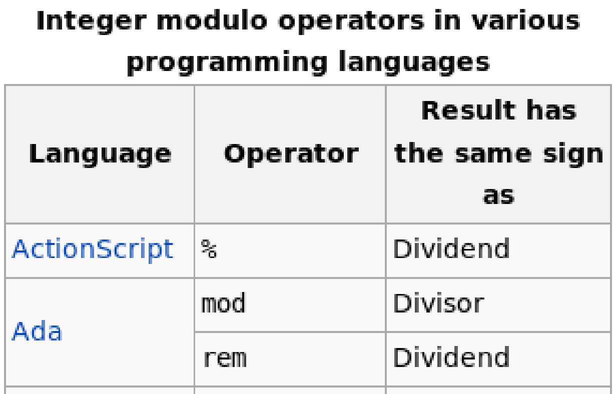
```
printf(“%d”, (-37 % 3));
```

results?

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

* If one operand is negative, result depends on the language. To check your language, consult this handy table**

** Wait, let's just never do that.



Integer modulo operators in various programming languages

Language	Operator	Result has the same sign as
ActionScript	%	Dividend
Ada	mod	Divisor
	rem	Dividend

Relational and Logical Operators

Used in evaluation conditions

```
if (expression evaluates to TRUE)
    ...do something...
```

What is TRUE (in C)?

- **0** means FALSE
- anything else (**1**, **-96**, **1.414**, **‘F’**, **inf**) means TRUE
- ???

```
float f = 9593.264;
if (f)
   ...do something...
```

Relational Operators

Six comparison operators: **<, >, ==, !=, >=, <=**

```
if (a < b) ...
if (x >= y) ...
if (q == r) ...
```

- Operands must be numbers (integer or floating point), result type is **`int`**
  - i.e., cannot use to compare structs, functions, arrays, etc.
- If relation is true, result is **1**, else result is **0**

```
float f = 9593.264;
if (f != 0)
    ...do something...
```

same meaning
as previous slide

Relational Operators (cont’d)

- Most common mistake in C (in my experience)
  - == is relational comparison for equality
  - = is assignment!

☠ *common source of bugs* ☠

confusion between
= and ==

Example: some strategic defense code…

```
if (enemy_launch = confirmed)
    retaliate();
```

Oops… sorry!

Logical Operators

Logical operators allow construction of complex (compound) conditions

Operands must be (or return) numbers (integer or floating point), result type is int

Logical NOT (!) operator

- result: 1 (TRUE) if operand was 0 (FALSE), otherwise 0

```
int j = ...;
if (! j)
   ... do something ...
```

```
float f = ..., g = ...;
if (! (f < g) )
   ... do something ...
```

Logical … (cont’d)

- AND (`&&`):
  - evaluate first operand, if 0, result is 0; else,
  - evaluate second operand, if 0, result is 0; else,
  - result is 1

```
if (x && (y > 32))
   ... do something ...
```

Logical… (cont’d)

- Condition evaluation stops as soon as truth value is known
  - i.e., order of the operands is significant
- Relied on by many programs!

☠ *common source of bugs* ☠

lack of understanding of significance of order in conditions

```
if ((b != 0) && ((a / b) > 5))
    printf(“quotient greater than 5\n”);
```

what’s the difference???

```
if (((a / b) > 5) && (b != 0))
    printf(“quotient greater than 5\n”);
```

Logical… (cont'd)

- OR (**||**) operator
  - evaluate first operand, if not **0**, result is **1**;
  - otherwise, evaluate second operand, if not **0**, result is **1**;
  - otherwise, result is **0**
- There is no logical XOR in C
  - but (**a** XOR **b**) → **`(a && (! b)) || ((! a) && b)`**

A Strange Idea?

- Mixing relational, bit-wise, and arithmetic operations into a single expression

```
unsigned char g, h;
int a, b;
float e, f;
…
if ((a < b) && (e * f || (g ^ h)))
    …do something here…
```

is condition true?

☠ *common source of bugs* ☠

mixing of operator types in a single expression

```
int a = -4;
char c = ‘D’;
float e = 0.0, f = 22.2, g;
…
g = (c == ‘D’) + (e || f) * a;
```

value of g?

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
```

Statement Termination and the “,”

- Normally, statements are executed sequentially and are separated by `;`
- Another separator: ‘,’ (e.g., `j = k++, i = k;`):
  1. evaluate expressions left to right
  2. complete all side effects of left expression before evaluating right expression
  3. result is value of the right expression
- More shorthand?

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

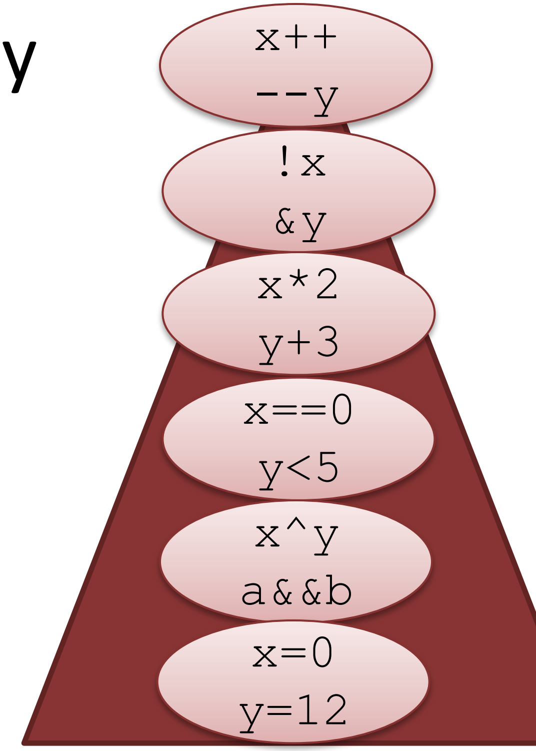
Tokens	Operator	Class	Prec.	Associates
`a[k]`	subscripting	postfix	16	left-to-right
`f(...)`	function call	postfix		left-to-right
`.`	direct selection	postfix		left-to-right
`->`	indirect selection	postfix		left to right
`++ --`	increment, decrement	postfix		left-to-right
`++ --`	increment, decrement	prefix	15	right-to-left
`sizeof`	size	unary		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 (dereference)	unary		right-to-left

C Operator Precedence (cont'd)

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

Precedence rules of thumb

- **Increment/decrement** are ultra sticky
- **Unary** operators are very sticky
- **Math is math**, and it’s pretty sticky
- **Comparisons** are not very sticky
- **Bitwise and logic** are very unsticky
- **Assignment** is positively repellant



x++
--y

!x
&y

x*2
y+3

x==0
y<5

x^y
a&&b

x=0
y=12

- Anything else? Not sure? USE PARENTHESES!!!!
  – Parentheses never hurt!!!

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?
- Ex:

```
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;
```

Exercise 05a

Operators

1. What does the following output?

```
int a = 32, b = 5;
printf(“%d %d\n”, a--, ++b);
printf(“%d %d\n”, --a, --b);
```

2. What is the value of a after executing the following, and is the condition TRUE or FALSE?

```
int a = 32, b = 5, c = 8, d = 4, e = 12;
if (a -= ((b > c) || (e / d)) + 6)
    ...do something...
```

Reminder: Go to course web page for link to exercise form.
Paste code into ideone.com and submit the link.

Flow of control

- Flow-of-control statements in C
  - `if-then-else`
  - `while` and `do-while`
  - `for`
  - `continue` and `break`
  - `switch-case`
  - `goto`
  - `conditional operator (?:)`
- Same set in java, except for `goto`
  - Which is bad anyway

    (unless you're a super kernel hacker, then go nuts)

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

Combining Assignment and Condition Checking

Why write this…

```
c = getchar();
while (c != ‘\n’) {
    …do something…
    c = getchar();
}
```

does the same thing!

…when you can write this instead?

```
while ( (c = getchar()) != ‘\n’ ) {
    …do something…
}
```

for

- The value of the counter after the loop is exited is valid and can be tested or used
  - C99: you can declare your counter in the for loop

```
for ( i = 0; i < 10; i++ )
   b *= 2;
printf("b was doubled %d times\n", i);
```

- Some parts of the expression can be missing; default to null statement

no initialization, i's value determined before the loop is executed

```
for (  ; i < 10; i++ )
   b *= 2;
```

break Statement

- Terminates execution of closest enclosing **for**, **while**, **do**, or **switch** statement

```
b = 0;
for ( i = 0; i < 10; i++ ) {
  for (j = 0; j < 5; j++) {
    if (a[i][j] > 100)
      break;
    b += a[j];
  }
  printf(“b = %d\n”, b);
}
```

which loop(s) does this exit?

Unlike Java, there is no labeled **break**

See: http://download.oracle.com/javase/tutorial/java/nutsandbolts/branch.html for example of a labeled break in Java.

continue Statement

- For bypassing 1 iteration of the innermost loop
  - but not exiting the loop altogether
- Example

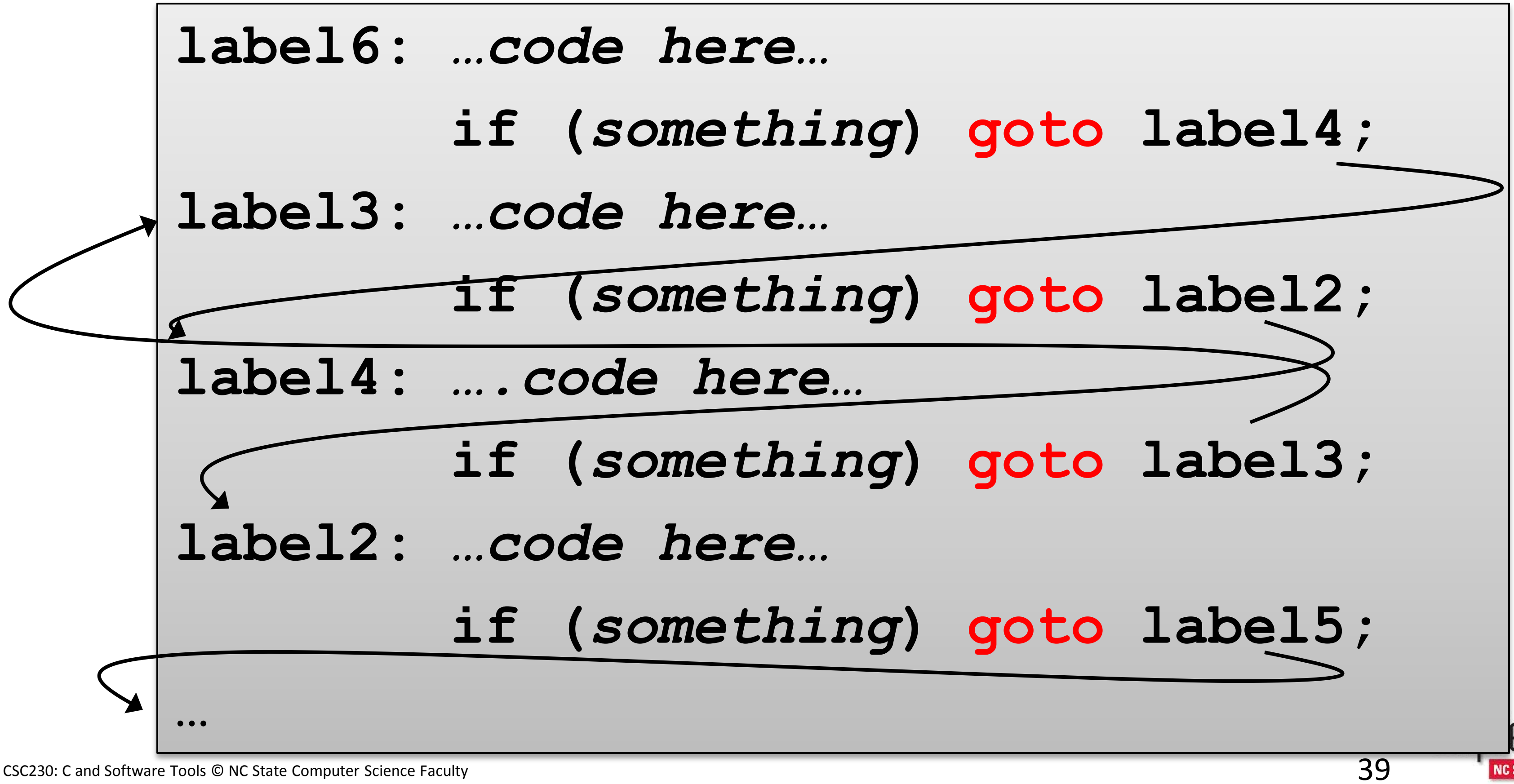
```
b = 0;
for ( i = 0; i < 10; i++ ) {
   for (j = 0; j < 5; j++) {
      if (a[i][j] > 100)
         continue;
      b += a[i][j];
   }
   printf(“b = %d\n”, b);
}
```

The `goto`

- Add symbolic labels (`thisisalabel:`)to arbitrary points in your program
- `goto <label>;` transfers control to that point

goto… (cont’d)

- General consensus: avoid using **goto’s**



```
label6: …code here…
        if (something) goto label4;
label3: …code here…
        if (something) goto label2;
label4: ….code here…
        if (something) goto label3;
label2: …code here…
        if (something) goto label5;
…
```

goto… (cont’d)

- Common exception: use for global exits (program termination)

```
for (…)
  for (…)
    for (…) {
      …
      if (disaster)
        goto whoops;
    }
…
whoops:
  /* clean up the mess here
    and abort execution */
```

Exercise 05b

Control flow

1. What are **d** and **g** equal to after…

```
int d=11, g=12;
int e=13, f=14;
int h=15;
int a = 2, b = 3;
int x = 40, y = 30;
if (a < b)
{
    d = e;
if (x > y)
    g = h;
}
else
    d = f;
```

2. Write an equivalent switch statement

```
unsigned int a;...
if ((a > 1) && (a <= 3))
    printf(“process now\n”);
else if (a == 5)
    printf(“defer til later\n”);
else if (a < 7)
    ;
else
    printf(“invalid code\n”);
```

Reminder: Go to course web page for link to exercise form.
Paste code into ideone.com and submit the link.