C Expressions, Operators, and Selection Statements

ITCS 2116: C Programming

College of Computing and Informatics

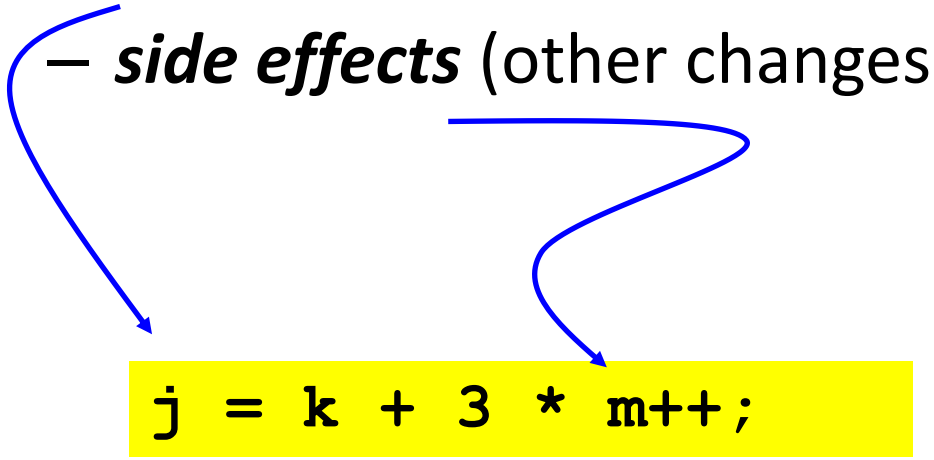
Department of Computer Science

Outline

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

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)



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?

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

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

Arithmetic on Two Operands

- Multiplication (`*`), Quotient (`/`), Remainder (`%`), Addition (`+`), Subtraction (`-`)
  - Possibility of underflow and overflow during expression evaluation, or assignment of the results

☠ *common source of bugs* ☠
overflow in computations

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

- 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* ☠
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 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
```

Flow of Control or Conditionals

Flow of control

- Flow-of-control statements in C
  - **`if-then-else`**
  - **`conditional operator (? : )`**
  - **`switch-case`**
  - **`for`**
  - **`continue and break`**
  - **`while`** and **`do-while`**

The `if` statement

- Allows a program to choose between two alternatives by evaluating an expression.
- Syntax:

```
if (expression) statement
```

- Example:

```
if (grade > 95)
   printf("A+");
```

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 in previous slide

Relational Operators

- C's ***relational operators:***

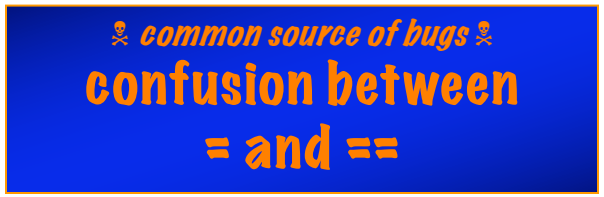
  `<` less than
  `>` greater than
  `<=` less than or equal to
  `>=` greater than or equal to

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

- Produce `0` (false) or `1` (true) when used in expressions.
- Can be used to compare integers and floating-point numbers, with operands of mixed types allowed.

Relational Operators (cont'd)

- **One of the most common mistakes in C**
  - **==** **is relational comparison for equality**
  - **=** is assignment!



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**, short-circuit evaluation
  - 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

  (**a** XOR **b**) ➔ `(a && (! b)) || ((!a) && b)`

The `else` clause

- **`if`** statements can have an else clause.
- The statement that follows **`else`** is executed if the expression evaluates to zero (*false*).
- Syntax:

```
if (expression) statement
else statement
```

The `else` clause: Example

```
if (age > 16)
  printf("Can drive");
else
  printf("Too young to drive");
```

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

Using Compound Statements

- Any group of statements that is surrounded by braces will be handled by the C compiler as a single statement.
- Syntax:

```
{
    statement1;
    statement2;
    …
    statementn;
}
```

Compound Statement Example

```
if (age > 16)
  printf("Can drive");
else
{
  printf("Too young to drive");
  printf("Please re-apply later");
}
```

Cascaded `if` statements

```
if (expression)
   statement
else if (expression)
   statement

…
else if (expression)
   statement
else
   statement
```

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

Cascaded `if` Statements

- A "cascaded" **`if`** statement is often the best way to test a series of conditions, stopping as soon as one of them is true.
- Example:

```
if (n < 0)
  printf("n is less than 0\n");
else
  if (n == 0)
    printf("n is equal to 0\n");
  else
    printf("n is greater than 0\n");
```

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

Cascaded `if` Statements

- Although the second **`if`** statement is nested inside the first, C programmers don't usually indent it.
- Instead, they align each **`else`** with the original **`if`**:

```
if (n < 0)
  printf("n is less than 0\n");
else if (n == 0)
  printf("n is equal to 0\n");
else
  printf("n is greater than 0\n");
```

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

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

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

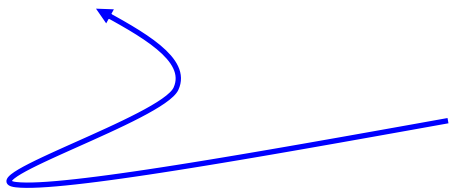
```
static int a = 35 + (16 % (4 | 1));
```

*(**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

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

The `switch` Statement

- A cascaded **`if`** statement can be used to compare an expression against a series of values:

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf(”Invalid grade");
```

The `switch` Statement (cont'd)

- The `switch` statement is an alternative:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Invalid grade");
           break;
}
```

The `switch` Statement (cont'd)

- A `switch` statement may be easier to read than a cascaded `if` statement.
- `switch` statements are often faster than `if` statements.
- Most common form of the `switch` statement:

```
switch ( expression ) {
   case constant-expression  :  statements
   …
   case constant-expression  :  statements
   default  :  statements
}
```

The `switch` Statement (cont'd)

- The word `switch` must be followed by an integer expression—the ***controlling expression***—in parentheses.
- Characters are treated as integers in C and thus can be tested in `switch` statements.
- Floating-point numbers and strings don't qualify, however.

The `switch` Statement (cont'd)

- Each case begins with a label of the form

  `case` *constant-expression* `:`

- A ***constant expression*** is much like an ordinary expression except that it cannot contain variables or function calls.
  - `5` is a constant expression, and `5 + 10` is a constant expression, but `n + 10` isn't a constant expression (unless `n` is a macro that represents a constant).
- The constant expression in a case label must evaluate to an integer (characters are valid).

The `switch` Statement (cont'd)

- After each case label comes any number of statements.
- No braces are required around the statements.
- The last statement in each group is normally `break`.

The `switch` Statement (cont'd)

- Duplicate case labels aren't allowed.
- The order of the cases doesn't matter, and the `default` case doesn't need to come last.
- Several case labels may precede a group of statements:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:  printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Invalid grade");
           break;
}
```

The `switch` Statement (cont'd)

- To save space, several case labels can be put on the same line:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          printf("Passing");
          break;
  case 0:  printf("Failing");
          break;
  default: printf("Invalid grade");
          break;
}
```

- If the **`default`** case is missing and the controlling expression's value doesn't match any case label, control passes to the next statement after the **`switch`**.

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

The Role of the `break` Statement

- Executing a **`break`** statement causes the program to "break" out of the **`switch`** statement; execution continues at the next statement after the **`switch`**.
- The **`switch`** statement is really a form of "computed jump."
- When the controlling expression is evaluated, control jumps to the case label matching the value of the **`switch`** expression.
- A case label is nothing more than a marker indicating a position within the **`switch`**.

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

The Role of the `break` Statement (cont'd)

- Without **`break`** (or some other jump statement) at the end of a case, control will flow into the next case.
- Example:

```
switch (grade) {
  case 4:  printf("Excellent");
  case 3:  printf("Good");
  case 2:  printf("Average");
  case 1:  printf("Poor");
  case 0:  printf("Failing");
  default: printf("Invalid grade");
}
```

- If the value of `grade` is 3, the message printed is

```
GoodAveragePoorFailingInvalid grade
```

The Role of the `break` Statement (cont'd)

- Omitting **`break`** is sometimes done intentionally, but it's usually just an oversight.
- It's a good idea to point out deliberate omissions of **`break`**:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          num_passing++;
          /* FALL THROUGH */
  case 0: total_grades++;
          break;
}
```

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

- Although the last case never needs a **`break`** statement, including one makes it easy to add cases in the future.

A Strange (or bad) Idea?

- Mixing relational, bit-wise, and arithmetic operations into a single expression. Is possible in C, but it is not recommended.

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

is condition true?

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

☠ *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;
```

What is the value of *g*?

References

- 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.