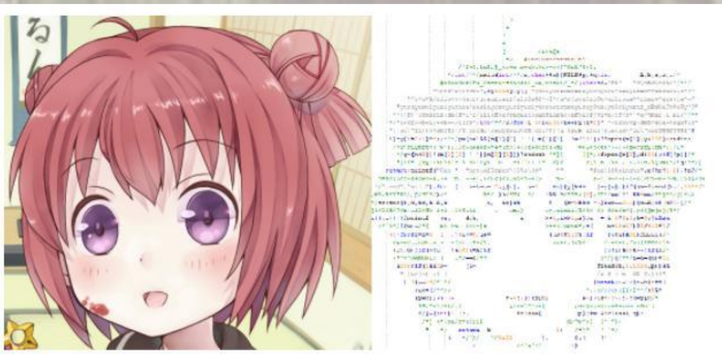
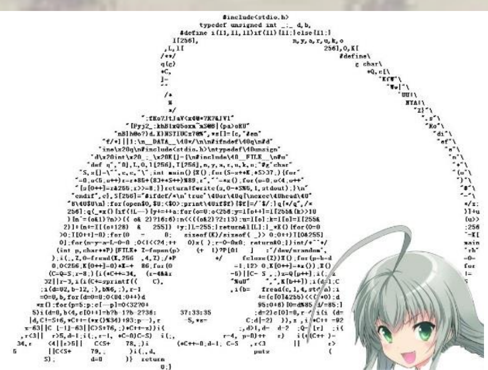
Lecture 4 - Expressions

Meng-Hsun Tsai
CSIE, NCKU





Operators

- C emphasizes expressions rather than statements.
- Expressions are built from variables, constants, and operators.
- C has a rich collection of operators, including
  - arithmetic operators
  - relational operators
  - logical operators
  - assignment operators
  - increment and decrement operators

  and many others

Arithmetic Operators

- C provides five binary ***arithmetic operators:***
  `+` addition
  `-` subtraction
  `*` multiplication
  `/` division
  `%` remainder
- An operator is ***binary*** if it has two operands.
- There are also two ***unary*** arithmetic operators:
  `+` unary plus
  `-` unary minus

Unary Arithmetic Operators

- The unary operators require one operand:

```
i = +1;
j = -i;
```

- The unary + operator does nothing. It’s used primarily to emphasize that a numeric constant is positive.

Binary Arithmetic Operators

- The value of `i % j` is the remainder when `i` is divided by `j`.

  `10 % 3` has the value 1, and `12 % 4` has the value 0.

- Binary arithmetic operators—with the exception of `%`—allow either integer or floating-point operands, with mixing allowed.
- When `int` and `float` operands are mixed, the result has type `float`.

  `9 + 2.5f` has the value 11.5, and `6.7f / 2` has the value 3.35.

IMSLab Since 2010

The / and % Operators

- The / and % operators require special care:
  - When both operands are integers, / "truncates" the result. The value of 1 / 2 is 0, not 0.5.
  - The % operator requires integer operands; if either operand is not an integer, the program won't compile.

```
x = 10 % 3.0;
```

```
error：invalid operands to binary % (have 「int」 and 「double」)
  x = 10 % 3.0;
         ^
```

The / and % Operators (cont.)

- Using zero as the right operand of either / or % causes undefined behavior.
  - Division by zero using integer arithmetic typically causes a program to terminate prematurely.
  - In floating-point arithmetic, some implementations allow division by zero, in which case positive or negative infinity is displayed as INF or -INF, respectively.

```
printf("2.0 / 0 = %f\n", 2.0/0);
printf("2 / 0 = %d\n", 2/0);
```

```
$ ./division_and_remainder
2.0 / 0 = inf
Floating exception (core dumped)
```

The `/` and `%` Operators (cont.)

- The behavior when `/` and `%` are used with negative operands is
  - ***implementation-defined*** in C89.
  - always truncated toward zero and the value of `i % j` has the same sign as `i` in C99.

```
printf("%d\n", 7 % 3);
printf("%d\n", 7 % -3);
printf("%d\n", -7 % 3);
printf("%d\n", -7 % -3);
```

```
1
1
-1
-1
```

Implementation-Defined Behavior

- The C standard deliberately leaves parts of the language unspecified.
- Leaving parts of the language unspecified reflects C’s emphasis on efficiency, which often means matching the way that hardware behaves.
- It’s best to avoid writing programs that depend on implementation-defined behavior.

Operator Precedence

- Does `i + j * k` mean "add `i` and `j`, then multiply the result by `k`" or "multiply `j` and `k`, then add `i`"?
- One solution to this problem is to add parentheses, writing either `(i + j) * k` or `i + (j * k)`.
- If the parentheses are omitted, C uses ***operator precedence*** rules to determine the meaning of the expression.

Operator Precedence (cont.)

- The arithmetic operators have the following relative precedence:

  Highest: `+` `-` (unary)
  `*` `/` `%`
  Lowest: `+` `-` (binary)

- Examples:

  `i + j * k` is equivalent to `i + (j * k)`

  `-i * -j` is equivalent to `(-i) * (-j)`

  `+i + j / k` is equivalent to `(+i) + (j / k)`

Operator Associativity

- ***Associativity*** comes into play when an expression contains two or more operators with equal precedence.
- An operator is said to be ***left associative*** if it groups from left to right.
- The binary arithmetic operators (`*`, `/`, `%`, `+`, and `-`) are all left associative, so

  `i - j - k` is equivalent to `(i - j) - k`
  `i * j / k` is equivalent to `(i * j) / k`
- An operator is ***right associative*** if it groups from right to left.
- The unary arithmetic operators (`+` and `-`) are both right associative, so

  `- + i` is equivalent to `-(+i)`

Program: Computing a UPC Check Digit

- Most goods sold in U.S. and Canadian stores are marked with a Universal Product Code (UPC):
- Meaning of the digits underneath the bar code:
  - First digit: Type of item (0)
  - First group of five digits: Manufacturer (13800)
  - Second group of five digits: Product (including package size) (15173)
  - Final digit: Check digit (5), used to help identify an error in the preceding digits

0 13800 15173 5

IMSLab Since 2010

Program: Computing a UPC Check Digit (cont.)

- How to compute the check digit (e.g. 0 13800 15173 5):
  1. Add the first, third, fifth, seventh, ninth, and eleventh digits.
     0 + 3 + 0 + 1 + 1 + 3 = 8
  2. Add the second, fourth, sixth, eighth, and tenth digits.
     1 + 8 + 0 + 5 + 7 = 21
  3. Multiply the first sum by 3 and add it to the second sum.
     3 * 8 + 21 = 45
  4. Subtract 1 from the total.
     45 – 1 = 44
  5. Compute the remainder when the adjusted total is divided by 10. Subtract the remainder from 9.
     9 – 44 %10 = **5**

IMSLab Since 2010

Program: Computing a UPC Check Digit (cont.)

upc.c

```
#include <stdio.h>
int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
      first_sum, second_sum, total;

  printf("Enter the first (single) digit: ");
  scanf("%1d", &d);
  printf("Enter first group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
  printf("Enter second group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);
  first_sum = d + i2 + i4 + j1 + j3 + j5;
  second_sum = i1 + i3 + i5 + j2 + j4;
  total = 3 * first_sum + second_sum;

  printf("Check digit: %d\n", 9 - ((total - 1) % 10));
  return 0;
}
```

Program: Computing a UPC Check Digit (cont.)

- The `upc.c` program asks the user to enter the first 11 digits of a UPC, then displays the corresponding check digit:

```
Enter the first (single) digit: 0
Enter first group of five digits: 13800
Enter second group of five digits: 15173
Check digit: 5
```

- The program reads each digit group as five one-digit numbers.
- To read single digits, we use `scanf` with the `%1d` conversion specification.

Assignment Operators

- ***Simple assignment:*** used for storing a value into a variable
- ***Compound assignment:*** used for updating a value already stored in a variable

Simple Assignment

- The effect of the assignment *v* = *e* is to evaluate the expression *e* and copy its value into *v*.
- *e* can be a constant, a variable, or a more complicated expression:

```
i = 5;            /* i is now 5  */
j = i;            /* j is now 5  */
k = 10 * i + j;   /* k is now 55 */
```

Simple Assignment (cont.)

- If *v* and *e* don't have the same type, then the value of *e* is converted to the type of *v* as the assignment takes place:

```
int i;
float f;
i = 72.99f;    /* i is now 72 */
f = 136;       /* f is now 136.0 */
```

- In many programming languages, assignment is a statement; in C, however, assignment is an operator, just like `+`.
- The value of an assignment *v* = *e* is the value of *v* after the assignment.
  - The value of `i = 72.99f` is 72 (not 72.99).

Side Effects

- An operator that modifies one of its operands is said to have a ***side effect.***
- The simple assignment operator has a side effect: it modifies its left operand.
- Evaluating the expression `i = 0` produces the result 0 and—as a side effect—assigns 0 to `i`.
- Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

- The `=` operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```

Side Effects (cont.)

- Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;
f = i = 33.3f;
```

  `i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3).

- "Embedded assignments" can make programs hard to read:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k); /* prints "1 1 2" */
```

- They can also be a source of subtle bugs.

Lvalues

- The assignment operator requires an ***lvalue*** as its left operand.
- An lvalue represents an object stored in computer memory, not a constant or the result of a computation.
- Variables are lvalues; expressions such as `10` or `2 * i` are not.
- It's illegal to put any other kind of expression on the left side of an assignment expression:

```
12 = i;         /*** WRONG ***/
i + j = 0;      /*** WRONG ***/
-i = j;         /*** WRONG ***/
```

- The compiler will produce an error message such as *"invalid lvalue in assignment."*

IMSLab Since 2010

Compound Assignment

- Assignments that use the old value of a variable to compute its new value are common. Example:

```
i = i + 2;
```

- Using the `+=` compound assignment operator, we simply write:

```
i += 2;    /* same as i = i + 2; */
```

Compound Assignment (cont.)

- There are nine other compound assignment operators, including the following:

  $-= \quad *= \quad /= \quad \%=$

- All compound assignment operators work in much the same way:
  - v += e adds v to e, storing the result in v
  - v −= e subtracts e from v, storing the result in v
  - v *= e multiplies v by e, storing the result in v
  - v /= e divides v by e, storing the result in v
  - v %= e computes the remainder when v is divided by e, storing the result in v

Compound Assignment (cont.)

- *v* += *e* isn't "equivalent" to *v* = *v* + *e*.
- One problem is operator precedence: `i *= j + k` isn't the same as `i = i * j + k`.
- There are also rare cases in which *v* += *e* differs from *v* = *v* + *e* because *v* itself has a side effect.

```
a[i++]+= 2;
```

- Similar remarks apply to the other compound assignment operators.
- When using the compound assignment operators, be careful not to switch the two characters that make up the operator.

```
i =+ j
```

- Although `i =+ j` will compile, it is equivalent to `i = (+j)`, which merely copies the value of `j` into `i`.

Increment and Decrement Operators

- Two of the most common operations on a variable are "incrementing" (adding 1) and "decrementing" (subtracting 1):

```
i = i + 1;
j = j - 1;
```

- Incrementing and decrementing can be done using the compound assignment operators:

```
i += 1;
j -= 1;
```

- C provides special `++` (***increment***) and `--` (***decrement***) operators.

- They can be used as ***prefix*** operators (`++i` and `--i`) or ***postfix*** operators (`i++` and `i--`).

IMSLab Since 2010

Increment and Decrement Operators (cont.)

- Evaluating the expression `++i` (a “pre-increment”) yields `i + 1` and—as a side effect—increments `i`:

```
i = 1;
printf("i is %d\n", ++i);    /* prints "i is 2" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

- Evaluating the expression `i++` (a “post-increment”) produces the result `i`, but causes `i` to be incremented afterwards:

```
i = 1;
printf("i is %d\n", i++);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

Increment and Decrement Operators (cont.)

- `++i` means "increment `i` immediately," while `i++` means "use the old value of `i` for now, but increment `i` later."
- How much later? The C standard doesn't specify a precise time, but it's safe to assume that `i` will be incremented before the next statement is executed.

Increment and Decrement Operators (cont.)

- The `--` operator has similar properties:

```
i = 1;
printf("i is %d\n", --i);    /* prints "i is 0" */
printf("i is %d\n", i);      /* prints "i is 0" */
i = 1;
printf("i is %d\n", i--);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 0" */
```

Increment and Decrement Operators (cont.)

- When `++` or `--` is used more than once in the same expression, the result can often be hard to understand.
- Example:

```
i = 1;
j = 2;
k = ++i + j++;
```

The last statement is equivalent to

```
i = i + 1;
k = i + j;
j = j + 1;
```

The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.

Increment and Decrement Operators (cont.)

- In contrast, executing the statements

```
i = 1;
j = 2;
k = i++ + j++;
```

will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.

Expression Evaluation

Table of operators discussed so far:

Precedence	*Name*	*Symbol(s)*	*Associativity*
1	increment (postfix) decrement (postfix)	++ --	left
2	increment (prefix) decrement (prefix) unary plus unary minus	++ -- + -	right
3	multiplicative	* / %	left
4	additive	+ -	left
5	assignment	= *= /= %= += -=	right

Expression Evaluation (cont.)

- The table can be used to add parentheses to an expression that lacks them.
- Starting with the operator with highest precedence, put parentheses around the operator and its operands.
- Example:

```
a = b += c++ - d + --e / -f
```

	Precedence level
`a = b += (c++) - d + --e / -f`	1
`a = b += (c++) - d + (--e) / (-f)`	2
`a = b += (c++) - d + ((--e) / (-f))`	3
`a = b += (((c++) - d) + ((--e) / (-f)))`	4
`(a = (b += (((c++) - d) + ((--e) / (-f)))))`	5

Order of Subexpression Evaluation

- The value of an expression may depend on the order in which its subexpressions are evaluated.
- C doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical and, logical or, conditional, and comma operators).
- In the expression `(a + b) * (c - d)` we don't know whether `(a + b)` will be evaluated before `(c - d)`.

Order of Subexpression Evaluation (cont.)

- Most expressions have the same value regardless of the order in which their subexpressions are evaluated.
- However, this may not be true when a subexpression modifies one of its operands:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

- The effect of executing the second statement is undefined.
- Avoid writing expressions that access the value of a variable and also modify the variable elsewhere in the expression.
- Some compilers may produce a warning message such as *"operation on 'a' may be undefined"* when they encounter such an expression.

Order of Subexpression Evaluation (cont.)

- To prevent problems, it's a good idea to avoid using the assignment operators in subexpressions.
- Instead, use a series of separate assignments:

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

The value of `c` will always be 6.

Order of Subexpression Evaluation (cont.)

- Besides the assignment operators, the only operators that modify their operands are increment and decrement.
- When using these operators, be careful that an expression doesn't depend on a particular order of evaluation. Example:

```
i = 2;
j = i * i++;
```

- It's natural to assume that `j` is assigned 4. However, `j` could just as well be assigned 6 instead:

1. The second operand (the original value of i) is fetched, then `i` is incremented.
2. The first operand (the new value of `i`) is fetched.
3. The new and old values of `i` are multiplied, yielding 6.

Undefined Behavior

- Statements such as `c = (b = a + 2) - (a = 1);` and `j = i * i++;` cause ***undefined behavior.***
- Possible effects of undefined behavior:
  - The program may behave differently when compiled with different compilers.
  - The program may not compile in the first place.
  - If it compiles it may not run.
  - If it does run, the program may crash, behave erratically, or produce meaningless results.
- Undefined behavior should be avoided.

IMSLab Since 2010

Expression Statements

- C has the unusual rule that any expression can be used as a statement. Example:

  ```
  ++i;
  ```

  `i` is first incremented, then the new value of `i` is fetched but then discarded.

- Since its value is discarded, there's little point in using an expression as a statement unless the expression has a side effect:

  ```
  i = 1;          /* useful */
  i--;            /* useful */
  i * j - 1;      /* not useful */
  ```

Expression Statements (cont.)

- A slip of the finger can easily create a "do-nothing" expression statement.
- For example, instead of entering

```
i = j;
```

we might accidentally type

```
i + j;
```

- Some compilers can detect meaningless expression statements; you'll get a warning such as *"statement with no effect."*

A Quick Review to This Lecture

- Arithmetic Operators
  Unary: `+ -` Binary: `+ - * / %`
- When both operands are integers, `/` "truncates" the result. The value of `1 / 2` is 0, not 0.5.
- If either operand of the % operator is not an integer, the program won't compile.

```
x = 10 % 3.0;
```

- Using zero as the right operand of either `/` or `%` causes undefined behavior. (e.g., integer -> terminate, float -> return INF)

```
x = 2 / 0;
```

terminate

```
x = 2.0 / 0;
```

INF

- `/` and `%` are used with negative operands
  - always truncated toward zero and the value of `i % j` has the same sign as `i`.

```
 7 % 3 = 1      7 % -3 = 1
-7 % 3 = -1    -7 % -3 = -1
```

A Quick Review to This Lecture (cont.)

- Assignment Operators

  Simple: `=`   Compound: `+=  -=  *=  /=  %=`

- *v* = *e* evaluates the expression *e* and copies (or converts) its value into *v*.

- Operators that modifies one of its operands is said to have a ***side effect:***

  Assignment Operators / Increment and Decrement Operators

- Unexpected results in chained assignments with type conversion

```
f = i = 33.3f;    // (float -> int -> float)
```

- "Embedded assignments" makes programs hard to read:

```
k = 1 + (j = i);
```

A Quick Review to This Lecture (cont.)

- Lvalue: something (e.g., variables) that can be left operand of an assignment operator.
- Increment and Decrement Operators

  Prefix: `++i` `--i` Postfix: `i++` `i--`

Increment/decrement immediately	Increment/decrement before the next statement is executed.

- Using `++` or `--` more than once makes the program hard to understand.

```
k = ++i + j++;
```

A Quick Review to This Lecture (cont.)

- Expression Evaluation (see p.32 for Precedence and Associativity)
  - C doesn't define the order in which subexpressions are evaluated (exception: logical and, logical or, conditional, and comma operators).
  - Undefined Behavior

    ```
    c = (b = a + 2) - (a = 1);
    ```

    - accessing a variable and also modifying the variable
    - accessing an incremented/decremented variable twice

    ```
    j = i * i++;
    ```

- Expression Statement
  - C allows any expression to be used as a statement.

    ```
    ++i;
    ```

  - Use it only when the expression has a side effect