

From Last Week

```
#include <stdio.h>
int main(void)
{
    int i;
    float x;

    i = 40;
    x = 839.21f;
    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
    printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

    return 0;
}
```

- Output:
| 40 | 40 | 40 | 040 |
| 839.210 | 8.392e+02 | 839.21 |
- The `d` specifier is used to display an integer in decimal form.
- Conversion specifiers for floating-point numbers:
 - e — Exponential format.
 - f — “Fixed decimal” format.
 - g — Either exponential format or fixed decimal format, depending on the number’s size. *p* indicates the maximum number of significant digits to be displayed. The `g` conversion won’t show trailing zeros. If the number has no digits after the decimal point, `g` doesn’t display the decimal point.

Escape Sequences

- Escape sequences enable strings to contain control characters and special characters.
- A partial list of escape sequences:

Alert (bell)	\a
Backspace	\b
New line	\n
Horizontal tab	\t
"	\"
\	\\

The scanf Function

- `scanf` reads input according to a particular format.
- A `scanf` format string may contain both ordinary characters and conversion specifications.
- The conversions allowed with `scanf` are essentially the same as those used with `printf`.
- In many cases, a `scanf` format string will contain only conversion specifications:

```
int i, j;
float x, y;
scanf("%d%d%f%f", &i, &j, &x, &y);
```
- Sample input:

```
1 -20 .3 -4.0e3
```

`scanf` will assign 1, -20, 0.3, and -4000.0 to `i`, `j`, `x`, and `y`, respectively.
- When using `scanf`, must check that
 - the number of conversion specifications matches the number of input variables and
 - each conversion is appropriate for the corresponding variable.
- Another trap involves the `&` symbol, which normally precedes each variable in a `scanf` call.

How scanf Works

- `scanf` tries to match groups of input characters with conversion specifications in the format string.
- For each conversion specification, `scanf` tries to locate an item of the appropriate type in the input data, skipping blank space, new line, tabs if necessary.
- `scanf` then reads the item, stopping when it reaches a character that can’t belong to the item.

- The unary arithmetic operators (+ and -) are both right associative, so
 $- + i$ is equivalent to $-(+i)$

Assignment Operators

- Simple assignment:** used for storing a value into a variable
- Compound assignment:** used for updating a value already stored in a variable, e.g. $i = i + 2$;
- 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

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

```
int i;
float f;

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

- In C, assignment is an operator, not a statement.
Example: $i = (j = (k = 0))$;
- The $=$ operator is right associative. $i=j=k=0$;
- An assignment of the form $v = e$ is allowed wherever a value of type v would be permitted:

Example:

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

Output: 1 1 2.

Comment: Embedded assignments makes codes hard to read.

- 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 $2i$ are not.

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

Side Effects

- An operators 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.

Compound Assignment

- Assignments that use the old value of a variable to compute its new value are common.
- Compound assignment operators include:

$+=$ $-=$ $*=$ $/=$ $\%=$

- 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

Increment and Decrement Operators

- C has $++$ and $--$ operators.
The $++$ adds 1 to its operand. The $--$ subtracts 1.
- The increment and decrement operators are tricky:
 - prefix** operators: $++i$ (pre-increment) and $--i$
 - postfix** operator: $i++$ (post-increment) and $i--$
 - Have side effects: modify the values of operands.

Example:

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

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

Question:

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

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

Question:

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

The last statement is equivalent to

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

```
i = 1;          /* useful */
i--;            /* useful */
i * j - 1;      /* no effect */
```

- Some compilers can detect meaningless expression statements; you'll get a warning such as "*statement with no effect.*". For example, instead of entering `i = j;` we might accidentally type `i + j;`

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

Expression 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 $(c - d)$.
- 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);
```

- Operators that have side effects: the assignment operators, increment, and decrement.
- When using operators with side effects, do not use expressions that depend on a particular order of evaluation.

Expression Statements

- C has the unusual rule that any expression can be used as a statement;
- but some expression statements have no effect, e.g. `i+j;` there's little point in using an expression as a statement unless the expression has a side effect: