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.

- If the item was read successfully, scanf continues processing the rest of the format string.
- If not, scanf returns immediately.

- As it searches for a number, scanf ignores ***white-space characters***.

```
Example:
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- The numbers can be on one line or spread over lines:

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

- scanf sees a stream of characters (¤ represents new-line, and • represents the space character). scanf "peeks" at the final new-line without reading it.

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr (s=skipped;r=read)
```

Example:

- Sample input:
  `1-20.3-4.0e3¤`
- The call of scanf is the same as before:
  `scanf("%d%d%f%f", &i, &j, &x, &y);`
- Here's how scanf would process the new input:
  - %d. Stores 1 into i and puts the - character back.
  - %d. Stores –20 into j and puts the . character back.
  - %f. Stores 0.3 into x and puts the - character back.
  - %f. Stores –4.0 × 103 into y and puts the new-line character back.

Example:

```
scanf("%d/%d", &i, &j);
```

- If the input is •5/•96, scanf succeeds.
- If the input is •5•/•96, scanf fails, because the / in the format string doesn't match the space in the input. But `scanf("%d /%d", &i, &j)` succeeds.

Differences between printf and scanf:

- One common mistake is to put & in front of variables in a call of printf:
  `printf("%d %d\n", &i, &j);  /* WRONG */`
- Putting a new-line character at the end of a scanf format string is usually a bad idea.
- To scanf, a new-line character in a format string is equivalent to a space; cause scanf to advance to the next non-white-space character.

Chapter 4 Expressions

- Expressions are built from variables, constants, and operators.
- Operators in C includes
  - arithmetic operators
  - relational operators
  - logical operators
  - assignment operators
  - increment and decrement operators

  and many other operators.

Arithmetic Operators

- Five binary ***arithmetic operators:***

+	addition
-	subtraction
*	multiplication
/	division
%	remainder. Only for integers

- Two ***unary*** arithmetic operators:

+	unary plus. This operator does nothing.
-	unary minus

  Example:
  ```
  i = +i;
  j = -i;
  ```

- When int and float operands are mixed, the result has type float.
- When both operands are integers, / "truncates" the result. The value of 1 / 2 is 0.
- The behavior when / and % are used with negative operands is ***implementation-defined*** in C89.

  In C99, the result of a division is always truncated toward zero and the value of i % j has the same sign as i.

- C uses ***operator precedence*** rules.

  Precedence for arithmetic operators

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

- 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`

- 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, no 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 2 i 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;
```

Precedence	*Name*	*Symbol(s)*	*Associativity*
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:

```
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;`