Spring 2007 Handout 1

Methodological preliminary

Type every character of every example *exactly* the way I type it. Use uppercase and lowercase exactly as I do. Type each blank and tab exactly as I do. Don't confuse the lowercase **`l`** with the digit **`1`**, Don't confuse the uppercase **`O`** with the digit **`0`**, etc.

If you get syntax errors, you must compare every character that you typed with every character in the example. Scan from left to right and be methodical.

A simple C program: K&R pp. 5−7; King pp. 9−16

A *computer* is a machine that follows a list of instructions. A *program* is a file (i.e., a document) that contains the list of instructions that you put into the computer.

Create a file named **`hello.c`**. Don't type the line numbers or the blank immediately after each line number. For example, leave no space before the **`#`**. Use only lowercase, except in comments.

A C program is divided into sections called *functions.* Each function has a name: **`printf`**, **`sqrt`**, **`rand`**, etc. To give the function a piece of data to work with, write it in parentheses after the name of the function. This data is called the *argument* of the function:

```
printf("hello")
sqrt(2.0)
rand()
```

Always write the parentheses, even if the function takes no arguments. The parentheses indicate that the word in front of them is the name of a function, and that you want the computer to *call* (i.e., execute) the function.

```
1 /* Print the word "hello". */
2 #include <stdio.h>
3
4 main()
5 {
6     printf("hello\n");
7 }
```

```
hello
```

To avoid a warning on some systems, insert the word **`void`** at the start of line 4 before the word **`main`**. Or insert the word **`int`** there and insert the statement

```
return 0;
```

between lines 6 and 7.

Compile and run the program: K&R p. 6; King pp. 10−11

To compile this program with the ANSI C compiler on i5,

```
1$ gcc hello.c
```
Create an executable program named **`a.out`**.

```
2$ ls -l a.out
```
Verify that you created **`a.out`**.

```
3$ gcc -o hello hello.c
```
Better to create an executable program named **hello**.

```
4$ ls -l hello
```
Verify that you created **hello**.

```
5$ gcc -o hello hello.c 2> hello.err
```
If there are error messages, save them in a file.

```
6$ ls -l hello.err
```

```
7$ hello
```
Run the program; send the output on screen.

```
8$ hello > hello.out
```
Run it again; send the output to the file **hello.out** *instead.*

```
9$ lpr -Pedlab hello.c hello.out
```
Print program and output.

```
10$ rm hello.out
```
Remove the output file.

What happens if you omit the \n: K&R pp. 7−8; King p. 14

```
1 #include <stdio.h>
2
3 main()
4 {
5     printf("Once I built a railroad,\n");
6     printf("Made it run:\n");
7     printf("Made it race against time.\n");
8 }
```

```
Once I built a railroad,
Made it run:
Made it race against time.
```

```
1 #include <stdio.h>
2
3 main()
4 {
5     printf("Once I built a railroad,");
6     printf("Now it's done:");
7     printf("Brother, can you spare a dime?");
8 }
```

```
Once I built a railroad,Now it's done:Brother, can you spare a dime?
```

How not to print

```
    printf("hello/n");

    /* Departure from accepted programming practice: */
    printf("\nOnce I built a railroad,");
    printf("\nNow it's done:");
    printf("\nBrother, can you spare a dime?");
```

Never output a blank (or a tab) immediately before a newline—it accomplishes nothing.

```
    printf("hello \n");          /* bad */
    printf("hello\n");           /* good */
```

▼ **Homework 1.1: print a flag**

Write a program named **`flag.c`** to print a flag with ten **`printf`**'s, each ending with a **`\n`**:

```
* * * * * * ====================
 * * * * *  ====================
* * * * * * ====================
 * * * * *  ====================
* * * * * * ====================
==================================
==================================
=========E pluribus unum=========
==================================
==================================
```

▲

Declare integer variables

A *variable* is a container that can hold a number. Each variable has a name, called its *identifier.* You can create a variable by writing a *declaration* as in lines 5−6 below. Put the declarations immediately after the **`{`** and before the other statements (e.g., assignment statements, **`printf`**'s, etc.) See K&R p. 9; King pp. 16−17 for variables and declarations; K&R pp. 35−36, 192; King pp. 23−24, 29 for the rules for identifiers.

An *expression* is built out of one or more numbers and/or variables, joined together with **`+ - * / %`**, etc., and parentheses. (Numbers are called *constants*.) An expression may be written to the right of the **`=`** in an *assignment statement* (lines 17, 20, 23 below) or as the second, third, fourth, etc., argument of **`printf`**.

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9232/src/variable.c`

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int i = 10;           /* Write comments alongside declarations. */
 6     int j = 20, k = 30;   /* Can declare two variables in same statement. */
 7     const int m = 40;     /* Can't change the value of this variable. */
 8
 9     printf("The value of i is %d.\n", i);
10     printf("%d\n", i);
11
12     printf("The value of i is %d, j is %d, and k is %d.\n", i, j, k);
13     printf("%d %d %d\n", i, j, k);
14
15     printf("The value of i + j * k is %d.\n", i + j * k);
16
17     i = 20;          /* Can change the value of a variable. */
18     printf("The new value of i is %d.\n", i);
19
20     i = 10 * j + k;
21     printf("The new value of i is %d.\n", i);
22
23     i = i + 1;       /* Add 1 to the value of i. */
24     printf("The new value of i is %d.\n", i);
25
```

```
26      printf("I agree 100%%.\n");
27  }
```

Line 5 could be split into

```
    int i;              /* Create a variable i containing garbage. */
    i = 10;             /* Replace the garbage by 10. */
```

—but why would you want to? See K&R pp. 40, 85−6; King pp. 19−20.

```
The value of i is 10.                       line 9
10                                          line 10
The value of i is 10, j is 20, and k is 30.     line 12
10 20 30                                    line 13
The value of i + j * k is 610.              line 15
The new value of i is 20.                   line 18
The new value of i is 430.                  line 21
The new value of i is 431.                  line 24
I agree 100%.                               line 26
```

A variable with an unpredictable value

For now, assume that every variable is born containing an unpredictable number. The following program may print a different value each time you run it. Or it may always print the same value. And it may behave differently on a different machine. There will be no error message.

```
1  #include <stdio.h>
2
3  main()
4  {
5      int i;
6
7      printf("%d\n", i);
8  }
```

Operator precedence: K&R pp. 52−54; King pp. 46−48, 595

For now, ignore lines 2 and 13 of the table on K&R p. 53; King p. 595. (Line 2 lists all the unary operators such as **++**; line 13 lists the ternary operator **?:**.) The table shows that ***** has higher precedence than **+**. To override this, parenthesize the operator of lower precedence:

```
    k = 1 + 2 * 3;          /* multiply before add: put 7 into k */
    k = (1 + 2) * 3;        /* add before multiply: put 9 into k */
```

Examples of / and %: K&R pp. 41, 205; King p. 46

The quotient of two integers will always be an integer, i.e., it will have no fractional part. Each machine reacts differently when you divide by zero (K&R p. 200, §A7). The result of **%**'ing by 5 will always be in the range 0−4 inclusive.

```
1  #include <stdio.h>
2
3  main()
4  {
5      printf("%d\n", 38 / 5);      /* quotient, truncated to integer */
6
7      printf("%d\n", 38 % 5);      /* remainder */
8      printf("%d\n", 39 % 5);
```

```
 9     printf("%d\n", 40 % 5);
10 }
```

```
7
3       because 38 is 3 more than a multiple of 5
4       because 39 is 4 more than a multiple of 5
0       because 40 is a multiple of 5
```

Other conversion characters for integer expressions

Surprisingly, **printf** has no **%b** for binary. See K&R pp. 11, 13. For the complete list, see K&R pp. 154, 244; King p. 489.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int i = 74;
 6
 7     printf("%d\n", i);       /* decimal */
 8     printf("%o\n", i);       /* octal */
 9     printf("%x\n", i);       /* lowercase hexadecimal */
10     printf("%X\n", i);       /* uppercase hexadecimal */
11     printf("%c\n", i);       /* ASCII character */
12 }
```

```
74
112
4a
4A
J
```

▼ Homework 1.2: discover some ASCII codes

I just showed you how to discover that uppercase **J** is the character whose ASCII code is decimal 74. Discover the characters whose ASCII codes are 65, 97, 48, 32. The last two will be tricky.

▲

Minimum field width

%5d will print blanks in front of the number, if necessary, so that the total width of the blanks and the number will be at least 5 characters. This will line up, or *right justify,* the numbers. **%5d** will never chop digits off the number, even if the number has more than five digits. See K&R p. 11, and the second bullet on K&R pp. 153, 243 or King pp. 488, 491.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int i = 6;               /* the first four perfect numbers */
 6     int j = 28;
 7     int k = 496;
 8     int l = 8128;
 9
10     printf("%d\n", i);
11     printf("%d\n", j);
```

```
12     printf("%d\n", k);
13     printf("%d\n", l);
14
15     printf("\n");            /* Print an empty line. */
16
17     printf("%5d\n", i);
18     printf("%5d\n", j);
19     printf("%5d\n", k);
20     printf("%5d\n", l);
21
22     printf("\n");            /* Print an empty line. */
23
24     printf("%3d\n", l);
25 }
```

```
6
28
496
8128
```

```
     6          four blanks before the 6
    28          three blanks before the 28
   496          two blanks before the 496
  8128          one blank before the 8128

8128            no blanks before the 8128, but no digits are chopped off.
```

Combine lines 13−15 to

```
printf("%d\n\n", l);
```

How not to print integer variables

%d always prints one or more digits. The following **1** is therefore redundant, since it asks for at least one character of output.

```
printf("%1d\n", i);          /* The digit 1 does nothing. */

printf("           %d           %d           %d\n", i, j, k);     /* bad */
printf("%12d %12d %12d\n", i, j, k);                              /* good */
```

Pad with zeroes instead of blanks: K&R p. 243 last sentence; King p. 488, 490.

The only padding characters that **printf** provides are blank and zero.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int agent = 7;
 6
 7     printf("%d\n", agent);
 8     printf("%3d\n", agent);
 9     printf("%03d\n", agent);
10 }
```

```
7               nothing before the 7
  7             two blanks before the 7
007             two zeroes before the 7
```

Input an integer: K&R pp. 93–94, 157; King pp. 20, 36–38

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int years;
 6
 7     printf("How old are you? ");          /* no newline */
 8     scanf("%d", &years);
 9     printf("You'll be ready to retire in %d years.\n", 65 - years);
10 }
```

```
How old are you? 52
You'll be ready to retire in 13years.
```

To print the value of a variable, you must give its current value to **`printf`**:

```
    printf("%d", years);
```

To give a new value to a variable, however, there is no need to give its current value to **`scanf`**. **`scanf`** doesn't care about the current value: **`scanf`**'s job is to destroy the current value and replace it by a new one. That's why we do not give **`years`** to **`scanf`** as its second argument.

To do its job of installing a new value into the variable, **`scanf`** needs a different kind of information: it needs to know the memory address of the variable. The value of the expression **`&years`** is the *address* of the variable, as opposed to its *value:*

```
    scanf("%d", &years);
```

How not to input an integer

It would be wasted effort to assign a value to **`years`** before the **`scanf`** in the above program. For example, don't say

```
    int years = 0;

    printf("How old are you? ");
    scanf("%d", &years);
```

▼ Homework 1.3: write a program that accepts input

Write a program named **`input.c`** that asks the user to type in one or more numbers, and then prints an answer. Any one of the following will do: you get no credit if you hand in more than one.

Use as few variables as possible. Don't worry about the difference between singular and plural: i.e., don't use **`if`**. Don't use **`while`** or **`for`** either. Do not write a minimum field width number between the **`%`** and the conversion character **`d`**.

```
How old are you? 52
That's 14049dog years.
```

```
What year is this? 2007
Then this Congress is the 110the Congress
```

Use **/** and **%**. Do not use multiplication:

```
How many minutes have you been waiting? 200
That's 3 hours and 20 minutes!
```

```
How many cents do you have? 239
That's 2 dollars and 39 cents!
```

```
How many pounds does the turkey weigh? 11
How many minutes does each pound take? 20
Then you have to cook it for 3 hours and 40 minutes.
```

You can do this two ways: 365 / 100 * 100, or 365 − 365 % 100. The latter is faster:

```
Please type a number: 365
Rounded down to the nearest 100, that's 300.
```

Later in the course, we will rewrite this program with an array of structures:

```
How many quarters do you have? 3
How many dimes do you have? 3
How many nickels do you have? 0
How many pennies do you have? 4
Then you have 1 dollars and 9 cents.
```

▲

How not to program

```
 1 /* Print the numbers from 1 to 10, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     printf("%2d\n",  1);
 7     printf("%2d\n",  2);
 8     printf("%2d\n",  3);
 9     printf("%2d\n",  4);
10     printf("%2d\n",  5);
11     printf("%2d\n",  6);
12     printf("%2d\n",  7);
13     printf("%2d\n",  8);
14     printf("%2d\n",  9);
15     printf("%2d\n", 10);
16
17     printf("That's all, folks!\n");
18 }
```

```
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
That's all, folks!
```

Two ways to count from 1 to 10: K&R pp. 8−14, 60−62, 224; King pp. 86−87, 91−93

The *body* of a loop is the statement(s) that are executed repeatedly (lines 10−11 below). Enclose the body within **{**curly braces**}** shown on lines 9 and 12. Indent the body one tab stop farther than the surrounding lines. For example, lines 10−11 are indented one tab stop farther than lines 9 and 12.

The **{**curly braces**}** around the body of the **`while`** or **`for`** are optional when the body consists of only one statement, but are required when the body consists of two or more statements. *In this course, however, you will always write the curly braces or you will get no credit for your homework.* The body below consists of the two statements on lines 10−11. If the body consists of no statements, follow the example on pp. 35−36 in this Handout rather than K&R pp. 18−19 or King p. 102.

Each trip through the loop is called an *iteration.* The following loop iterates 10 times. The variable that gets a new value during each iteration is called the *induction variable* of the loop.

```
 1 /* Print the numbers from 1 to 10, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7
 8     i = 1;
 9     while (i <= 10) {                    /* no semicolon after the ) */
10         printf("%2d\n", i);
11         i = i + 1;
12     }
13     printf("That's all, folks!\n");
14 }
```

```
 1 /* A more localized notation for printing the numbers from 1 to 10, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7
 8     for (i = 1; i <= 10; i = i + 1) {     /* no semicolon after the ) */
 9         printf("%2d\n", i);
10     }
11     printf("That's all, folks!\n");
12 }
```

Here are the names of the parts of a **`for`** loop, and the order in which they're executed. Let's assume the loop iterates ten times.


```
1     for (initialization; test; re-initialization) {
2         body
3     }
```

1. initialization (executed only once. George Bush: ‘‘I was outside the loop.’’)
2. test
3. body
4. re-initialization
5. test
6. body
7. re-initialization
8. test
9. body
10. re-initialization

. . .

30. body
31. re-initialization
32. test (is always last)

```
 1 /* Print graph paper with 10 rows and 10 columns. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int row;
 7
 8     for (row = 1; row <= 10; row = row + 1) {
 9         printf("+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+\n");
10         printf("|     |     |     |     |     |     |     |     |     |     |\n");
11     }
12 }
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|     |     |     |     |     |     |     |     |     |     |
etc.
```

Nested loops

The **\n\n** prints an empty line after the **Aaaaaaaaaaaaaaaaaaaaaaaahhh**.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int i;
 6     int j;
 7
 8     for (i = 1; i <= 2; i = i + 1) {
 9         for (j = 1; j <= 3; j = j + 1) {
10             printf("Lucy in the sky with diamonds\n");
11         }
```

```
12          printf("Aaaaaaaaaaaaaaaaaaaaaaaahhh\n\n");
13      }
14  }
```

```
Lucy in the sky with diamonds
Lucy in the sky with diamonds
Lucy in the sky with diamonds
Aaaaaaaaaaaaaaaaaaaaaaaahhhh

Lucy in the sky with diamonds
Lucy in the sky with diamonds
Lucy in the sky with diamonds
Aaaaaaaaaaaaaaaaaaaaaaaahhhh
```

More nested loops

```
 1  #include <stdio.h>
 2
 3  main()
 4  {
 5      int j;
 6
 7      for (j = 1; j <= 5; j = j + 1) {     /* print a row of 5 X's */
 8          printf("X");
 9      }
10
11      printf("\n");
12  }
```

```
XXXXX
```

```
 1  #include <stdio.h>
 2
 3  main()
 4  {
 5      int row;
 6      int col;
 7
 8      for (row = 1; row <= 4; row = row + 1) {      /* print a 4 by 5   */
 9          for (col = 1; col <= 5; col = col + 1) {  /* rectangle of X's */
10              printf("X");
11          }
12          printf("\n");
13      }
14  }
```

```
XXXXX
XXXXX
XXXXX
XXXXX
```

Change **5** to **row** in the above line 9 to produce the following output:

```
X
XX
XXX
XXXX
```

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int row;
 6     int i;
 7
 8     for (row = 1; row <= 4; row = row + 1) {
 9         for (i = 1; i <= 4 - row; i = i + 1) {
10             printf(" ");
11         }
12
13         for (i = 1; i <= row; i = i + 1) {
14             printf("X");
15         }
16
17         printf("\n");
18     }
19 }
```

```
   X
  XX
 XXX
XXXX
```

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int row;
 6     int i;
 7
 8     for (row = 1; row <= 5; row = row + 1) {
 9         for (i = 1; i <= 5 - row; i = i + 1) {
10             printf(" ");
11         }
12
13         for (i = 1; i <= 2 * row - 1; i = i + 1) {
14             printf("X");
15         }
16
17         printf("\n");
18     }
19 }
```

```
    X
   XXX
  XXXXX
 XXXXXXX
XXXXXXXXX
```

Don't write the same number in more than one place.

To change the number of columns, we'd have to edit lines 10 and 15. Worse, it's hard to tell which **10**'s stand for the number of rows, and which **10**'s stand for the number of columns.

```
 1 /* Print graph paper with 10 rows and 10 columns. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int row;
 7     int col;
 8
 9     for (row = 1; row <= 10; ++row) {
10         for (col = 1; col <= 10; ++col) {
11             printf("+-----");
12         }
13         printf("\n");
14
15         for (col = 1; col <= 10; ++col) {
16             printf("|     ");
17         }
18         printf("\n");
19     }
20 }
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
|     |     |     |     |     |     |     |     |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
|     |     |     |     |     |     |     |     |     |
```
etc.

```
 1 /* Print graph paper with nrows rows and ncols columns. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     const int nrows  = 10;    /* number of rows */
 7     const int ncols  = 10;    /* number of columns */
 8
 9     int row;
10     int col;
11
12     for (row = 1; row <= nrows; ++row) {
13         for (col = 1; col <= ncols; ++col) {
14             printf("+-----");
15         }
```

```
16          printf("\n");
17
18          for (col = 1; col <= ncols; ++col) {
19              printf("|     ");
20          }
21          printf("\n");
22      }
23  }
```

#define a macro:
K&R pp. 14−15, 89−91, 229−231; King pp. 277−288

NROWS looks like a variable, but it's not. A **#define** line is merely an instruction to a word processor called the *C preprocessor,* asking it to change **NROWS** to **10** throughout the rest of the **.c** file. **NROWS** is called a *macro,* **10** is the *replacement text,* and the act of performing the change is called *macro expansion.*

The name of a macro should be all uppercase to remind you that it's not a variable. The replacement text can be more than a single number.

We can now change the number of columns by editing only line 5. A macro has been used correctly only if you can change the value without editing any line other than the **#define** line.

```
 1  /* Print graph paper with NROWS rows and NCOLS columns. */
 2  #include <stdio.h>
 3
 4  #define  NROWS   10      /* number of rows */
 5  #define  NCOLS   10      /* number of columns */
 6
 7  main()
 8  {
 9      int row;
10      int col;
11
12      for (row = 1; row <= NROWS; ++row) {
13          for (col = 1; col <= NCOLS; ++col) {
14              printf("+-----");
15          }
16          printf("\n");
17
18          for (col = 1; col <= NCOLS; ++col) {
19              printf("|     ");
20          }
21          printf("\n");
22      }
23  }
```

Macro definitions in the file /usr/include/stdlib.h:
K&R pp. 163−164, 252; King pp. 496−498

```
 1  /* Largest return value of rand, K&R p. 252. */
 2  #define RAND_MAX 32767
 3
 4  /* Return value of getchar (Handout 2, p. 21).
 5  There is no char with this value. */
 6  #define EOF (-1)
 7
```

```
 8 /* There is no variable at address NULL,
 9 so put NULL into a pointer to make sure it doesn't point at any variable. */
10 #define NULL 0
11
12 /* Arguments of exit. */
13 #define  EXIT_FAILURE 1
14 #define  EXIT_SUCCESS 0
```

▼ Homework 1.4: output a flag

Write a program with nested loops that produces the following output. The union jack is the rectangle in the upper left corner that contains the stars and blue background.

```
* * * ========
* * * ========
* * * ========
==============
==============
==============
```

Start the program with the following **`const int`**'s, which give the dimensions in characters:

```
1     const int height = 6;    /* height of whole flag */
2     const int uheight = 3;   /* height of union jack */
3
4     const int length = 14;   /* length of whole flag */
5     const int ulength = 6;   /* length of union jack */
```

The **`const int`**'s will determine how many times the loops iteratate. For example, the loop that prints the top stripe will iterate **`length - ulength`** times, and each time it will print the character **`"="`**. And the loop that prints the top row of stars will iterate **`ulength / 2`** times, and each time it will print the two characters **`"* "`**.

▲

Geometric progression in a for loop

```
 1 /* Print the powers of 2 from 1 to 64 inclusive, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7
 8     for (i = 1; i <= 64; i = i * 2) {
 9         printf("%2d\n", i);
10     }
11 }
```

```
 1
 2
 4
 8
16
32
64
```

Relational and logical operators: K&R pp. 21, 41−42; King pp. 64−66

Put no space between the two equal signs of a **==** etc. Be sure to type a double equal sign—a single equal sign means something else.

==	*equals*
!=	*not equals*
<	*less than*
<=	*less than or equals*
>	*greater than*
>=	*greater than or equals*
&&	*and*
||	*or*

For example,

```
i <= 10
i >= 1
i > 0

day >= 1 && day <= last
day < 1 || day > last
```

You get no error message for the following mistake; see K&R p. 19; King p. 79.

```
if (a = b) {          wrong
if (a == b) {         right
```

▼ Homework 1.5: A Hundred Bottles of Beer on the Wall

Write a program named **bottle.c** that will print the words to *A Hundred Bottles of Beer on the Wall*. It must have exactly one variable, one **for** loop, and four **printf**'s. Name the variable **b**. All four **printf**'s must be within the **for** loop. Do not write a minimum field width number between the **%** and the conversion character **d**. Print an empty line between verses by ending the last **printf** with **\n\n** instead of **\n**. Please hand in no more than one page of output.

For programs that print the words to songs, see Donald E. Knuth's article ''The Complexity of Songs'' in *SIGACT News,* Volume 9 Number 2 (Summer 1977), pp. 17−24.


```
100 bottles of beer on the wall,
100 bottles of beer--
If one of those bottles should happen to fall,
99 bottles of beer on the wall.

99 bottles of beer on the wall,
99 bottles of beer--
If one of those bottles should happen to fall,
98 bottles of beer on the wall.

98 bottles of beer on the wall,
98 bottles of beer--                    ad nauseam
```

▲

Never write a loop that always iterates exactly once

Never make a program more complicated than necessary. See William Strunk and E. B. White, *The Elements of Style, Third Edition,* p. 23, §17 (also pp. xiii−xiv).

Never write a loop that always iterates exactly once. Remove lines 8 and 10 (and 6 too) from each of the following two examples.

```
 1 /* Print the word "hello". */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7
 8     for (i = 1; i <= 1; i = i + 1) {
 9         printf("hello\n");
10     }
11 }
```

```
 1 /* Print the word "hello". */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7
 8     for (i = 1; i < 2; i = i + 1) {
 9         printf("hello\n");
10     }
11 }
```

Don't store a dead value in a variable

There is no need to put the value **1** into **i** twice.

```
 1 /* Print the numbers from 1 to 10, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i = 1;
```

```
 7
 8     for (i = 1; i <= 10; i = i + 1) {
 9         printf("%d\n", i);
10     }
11 }
```

Pick the right starting and ending numbers

```
 1 /* Print the numbers from 1 to 10, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7
 8     for (i = 2; i <= 11; i = i + 1) {
 9         printf("%d\n", i - 1);
10     }
11 }
```

Count in the right direction

```
 1 /* Print the numbers from 1 to 10, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7
 8     for (i = 10; i >= 1; i = i - 1) {
 9         printf("%d\n", 11 - i);
10     }
11 }
```

Don’t make a shadow variable

```
 1 /* Print the numbers from 1 to 10, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7     int j;
 8
 9     j = 0;
10     for (i = 1; i <= 10; i = i + 1) {
11         j = j + 1;
12         printf("%d\n", j);
13     }
14 }
```

Write all the repetitions inside the body of the loop

```
 1 /* Print the numbers from 1 to 10, one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i = 1;
 7
 8     printf("%d\n", i);
 9
10     for (i = 2; i <= 10; i = i + 1) {
11         printf("%d\n", i);
12     }
13 }
```

Rewrite the following program with no **printf**'s outside the **for** loop.

```
 1 /* Print A B A B A B A B etc., one per line. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     printf("A\n");
 7
 8     for (;;) {
 9         printf("B\n");
10         printf("A\n");
11     }
12 }
```

▼ Homework 1.6: eliminate repetitious code

Rewrite **pierogi.c** with a **for** loop that iterates seven times. The program must produce exactly the same output as the program below.

The program must have exactly one variable, one **for** loop, and two **printf**'s. Name the variable **h**: it must be the number of Hungarians, not the number of pierogies. One **printf** must be before the **for** loop and the other **printf** must be inside the **for** loop. You get no credit if you perform division.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     printf("This program helps you decide how many pierogies to make.\n\n");
 6
 7     printf("%2d pierogies feed %2d people or %2d Hungarians.\n",  8,  4, 2);
 8     printf("%2d pierogies feed %2d people or %2d Hungarians.\n", 12,  6, 3);
 9     printf("%2d pierogies feed %2d people or %2d Hungarians.\n", 16,  8, 4);
10     printf("%2d pierogies feed %2d people or %2d Hungarians.\n", 20, 10, 5);
11     printf("%2d pierogies feed %2d people or %2d Hungarians.\n", 24, 12, 6);
12     printf("%2d pierogies feed %2d people or %2d Hungarians.\n", 28, 14, 7);
13     printf("%2d pierogies feed %2d people or %2d Hungarians.\n", 32, 16, 8);
14 }
```

```
This program helps you decide how many pierogies to make.

 8 pierogies feed  4 people or  2 Hungarians.
12 pierogies feed  6 people or  3 Hungarians.
16 pierogies feed  8 people or  4 Hungarians.
20 pierogies feed 10 people or  5 Hungarians.
24 pierogies feed 12 people or  6 Hungarians.
28 pierogies feed 14 people or  7 Hungarians.
32 pierogies feed 16 people or  8 Hungarians.
```

▲

▼ Homework 1.7: the New York State Thruway

Write a program to output eight signs at 10 mile intervals in the following order (northbound). Output one empty line after each sign:

```
+--------------+
| Albany   108 |
| Montreal 328 |
| Buffalo  388 |
+--------------+

+--------------+
| Albany    98 |
| Montreal 318 |
| Buffalo  378 |
+--------------+

+--------------+
| Albany    88 |
| Montreal 308 |
| Buffalo  368 |
+--------------+
```

etc.

The program must contain exactly five **`printf`**'s (because the sign has five lines), one **`for`**, and one variable. Name the variable **`b`**: it must be the number of miles to Buffalo. Right-justify the numbers. You get no credit if you perform multiplication or division.

▲

▼ Homework 1.8: print a table

Write a program named **`table.c`** that will print a table with three columns listing the numbers from 0 to 31 inclusive in decimal, octal, and uppercase hexadecimal. You get no credit if you print in lowercase hexadecimal, or if you start or end at the wrong number. Print a heading on top of each column.

Unless you do the extra credit part, your program must have exactly one variable, one **`for`** loop, and two **`printf`**'s. One **`printf`** must be before the **`for`** loop and the other **`printf`** must be inside the **`for`** loop. The **`printf`** in the **`for`** loop will print three numbers each time it is executed. Right justify each column of numbers with **`%7`** before the conversion character.

The output should begin


```
decimal   octal     hex
      0       0       0
      1       1       1
      2       2       2
      3       3       3
      4       4       4
      5       5       5
      6       6       6
      7       7       7
      8      10       8
      9      11       9
     10      12       A       etc.
```

✎ Extra credit for space cadets only. Print a binary column in front of the decimal, octal, and hex columns. Each binary number will have five bits, ranging from **00000** to **11111**. For simplicity, always print all five bits:

```
binar decimal   octal     hex
00000       0       0       0
00001       1       1       1
00010       2       2       2
00011       3       3       3
00100       4       4       4       etc.
```

You get extra credit only if you solve this problem in the following way. Add a new **for** loop inside the existing **for** loop. The new **for** loop should iterate five times and its induction variable should count through the five numbers 16, 8, 4, 2, 1. The new **for** loop will contain exactly one **printf**, which will print exactly one digit (**0** or **1**) each time it is executed. The program must now have exactly three **printf**'s, two variables (which must both be **int**'s), and no **if**'s.

There must be exactly one pair of parentheses after each **printf**. You get no extra credit if there is a second pair of parentheses in any **printf** statement. The expression in the new **printf** should first use **/** to remove trailing digits from the number to be printed, and then use **%** to remove leading digits from the number. Don't write **/** and **%** more than once in the **printf**. You get no credit if you use **\b**, the **pow** function, or an array.

The folowing examples are in base 10. You will use base 2.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int n = 1234;
 6
 7     printf("%d\n", n / 1);          /* Remove trailing digit(s). */
 8     printf("%d\n", n / 10);
 9     printf("%d\n", n / 100);
10     printf("%d\n\n", n / 1000);
11
12     printf("%d\n", n % 1000);       /* Remove leading digit(s). */
13     printf("%d\n", n % 100);
14     printf("%d\n\n", n % 10);
15
16     printf("%d\n", n / 10 % 10);    /* Remove 1 trailing & all but 1 leading. */
17 }
```



```
1234
123
12
1

234
34
4

3
```

▲

Infinite for loop: K&R p. 60; King p. 94

On some machines (e.g., our Unix machine i5 at NYU), you may have to

```
#include <unistd.h>
```

after line 1 to use **`sleep`**. In Unix, type **`control-c`** to kill this program.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     for (;;) {
 6         printf("It was a dark and stormy night.\n");      /* Bulwer-Lytton */
 7         printf("Some Indians were sitting around a campfire.\n");
 8         printf("Then their chief rose and said:\n\n");
 9
10         sleep(5);        /* Do nothing for 5 seconds. */
11     }
12 }
```

```
It was a dark and stormy night.
Some Indians were sitting around a campfire.
Then their chief rose and said:

It was a dark and stormy night.
Some Indians were sitting around a campfire.
Then their chief rose and said:

It was a dark and stormy night.            etc.
```

Print random numbers: K&R pp. 251−252; King p. 570

`rand()` is an expression, just like **`i`** or **`i+j`** or **`2+2`**. Unlike **`2+2`**, however, the value of **`rand()`** will be different each time you use it. Like any other expression, it can appear in a **`printf`** or to the right of the **`=`** in an assignment statement.

It's easy to recognize a function in C: it always has parentheses after its name, even if it has no arguments:

```
printf("hello");
sqrt(2.0)
rand()
```

```
1 /* Print 10 random integers, one per line. */
```



```
 2 #include <stdio.h>
 3 #include <stdlib.h>      /* needed for rand */
 4
 5 main()
 6 {
 7     int i;
 8
 9     for (i = 1; i <= 10; i = i + 1) {
10         printf("%10d\n", rand());        /* percent ten d */
11     }
12 }
```

```
1103527590
 377401575
 662824084
1147902781
2035015474
 368800899
1508029952
 486256185
1062517886
 267834847
```

The numbers may be different on your machine.
one blank before the **377401575**

Print different random numbers each time you run the program: K&R pp. 252, 255−256; King pp. 89−91

If the above program prints the same ten random numbers each time you run it, call the **srand** function at the start of the program before doing anything else. Call **srand** only once—do not put it inside of a loop. **srand** does not return a random number, so don't attempt to **printf srand**. **srand** merely ensures that any subsequent call to **rand** will return a different random number each time you run the program.

```
 1 /* Print 10 random integers, one per line. */
 2
 3 #include <stdio.h>
 4 #include <stdlib.h>      /* needed for srand and rand */
 5 #include <time.h>        /* needed for time */
 6
 7 main()
 8 {
 9     int i;
10
11     srand(time(NULL));
12
13     for (i = 1; i <= 10; i = i + 1) {
14         printf("%10d\n", rand());
15     }
16 }
```

do-while loop: K&R pp. 63−64; King pp. 89−91

Unlike a **while** loop, a **do-while** loop will always iterate at least once.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
```

```
 3 #include <time.h>
 4
 5 main()
 6 {
 7     int dummy;
 8
 9     srand(time(NULL));
10
11     printf("Welcome to Russian Roulette.\n");
12     printf("Any number of players can take turns.\n");
13     printf("To pull the trigger when you see the ->,\n");
14     printf("type 0 and press RETURN.\n\n");
15
16     do {
17         printf("-> ");
18         scanf("%d", &dummy);
19     } while (rand() % 6 != 0);      /* one out of six chance of death */
20
21     printf("BANG!\n");
22 }
```

```
Welcome to Russian Roulette.
Any number of players can take turns.
To pull the trigger when you see the ->,
type 0 and press RETURN.

-> 0
-> 0
-> 0
BANG!
```

if statement: K&R p. 19; King pp. 66−68

The **(**parentheses**)**, **{**curly braces**}**, and indentation of an **`if`** are exactly like that of a **`while`**.

The *body* of an **`if`** is the statement(s) that may or may not be executed (lines 14−16 below). Enclose the body within **{**curly braces**}** (they're on lines 13 and 17). Indent the body one tab stop farther than the surrounding lines. For example, lines 14−16 are indented one tab stop farther than lines 13 and 17.

The **{**curly braces**}** around the body are optional when it consists of only one statement, but are required when it consists of two or more statements. *In this course, however, you will always write the curly braces or you will get no credit for your homework.* The body below consists of the three statements on lines 14−16.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int gore;
 6     int bush;
 7
 8     printf("How many electoral votes did Gore get? ");
 9     scanf("%d", &gore);
10
11     printf("How many electoral votes did Bush get? ");
12     scanf("%d", &bush);
```

```
13
14      if (gore == bush) {
15          printf("They're equal.\n");
16          printf("Please inform the House of Representatives\n");
17          printf("that the electoral college is hung.\n");
18      }
19  }
```

Never write an if that is always true or always false

Rearrange this program fragment, without changing its net effect.

```
1       if (a == a) {
2           b = c;
3       }
4
5       if (a != a) {
6           b = c;
7       }
```

What is the range of your rand function?

```
1  /* Output the largest of the first 1000 random integers returned by rand. */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  main()
8  {
9       int i;       /* loop counter */
10      int m;       /* largest random number seen so far */
11      int r;       /* each random number */
12
13      srand(time(NULL));
14      m = rand();
15
16      for (i = 1; i < 1000; i = i + 1) {
17          r = rand();
18          if (r > m) {
19              m = r;
20          }
21      }
22
23      printf("%d\n", m);
24  }
```

Can you write a program that will print the smallest of the first 1000 random numbers? Don't forget to change the comments.

if-then-else statement: K&R pp. 20−21, 55−58; King pp. 68−69

You will often need a pair of consecutive **`if`**'s of which exactly one will always be true: never both, and never neither.

```
1       if (a == b) {
2           printf("equal");
```



```
3     }
4
5     if (a != b) {
6         printf("not equal");
7     }
```

Use the word **`else`** (which means ‘‘otherwise’’) to make the program faster and smaller:

```
1     if (a == b) {
2         printf("equal");
3     } else {
4         printf("not equal");
5     }
```

The ‘‘then’’ section and the ‘‘else’’ section can each contain more than one statement. The same punctuation and indentation rules apply to both sections:

```
1     if (gore == bush) {
2         printf("They're equal.\n");
3         printf("Please inform the House of Representatives\n");
4         printf("that the electoral college is hung.\n");
5     } else {
6         printf("There's a clear winner.\n");
7         printf("Thank you, electors.\n");
8     }
```

▼ Homework 1.9: which of these can you combine using `else`?

You can use **`else`** only when you have a pair of consecutive **`if`**’s of which exactly one will always be true: never both, and never neither.

```
1     if (a < b) {
2         printf("less than");
3     }
4
5     if (a >= b) {
6         printf("greater than or equal");
7     }

8     if (a < b) {
9         printf("less than");
10    }
11
12    if (a > b) {
13        printf("greater than");
14    }

15    if (a <= b) {
16        printf("less than or equal");
17    }
18
19    if (a >= b) {
20        printf("greater than or equal");
21    }
```

▲

How they programmed before they invented else

The following example shows how they programmed before they invented **`else`**. Every time this program runs, the assignment to **`pay`** on line 12 is executed. Fifty percent of the time the assignment to **`pay`** on line 14 is also executed. Fix it so that only one assignment to **`pay`** is executed during each run.

```
 1 /* Flip a coin to decide somebody's salary: either $34 or $39 per hour. */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <time.h>
 5
 6 main()
 7 {
 8     int pay;
 9
10     srand(time(NULL));
11
12     pay = 34;
13     if (rand() % 2 == 0) {          /* fifty-fifty chance */
14         pay = 39;
15     }
16
17     printf("$%d per hour\n", pay);
18 }
```

```
 1 /* Flip a coin to decide somebody's salary: either $34 or $39 per hour. */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <time.h>
 5
 6 main()
 7 {
 8     int pay;
 9
10     srand(time(NULL));
11
12     if (rand() % 2 == 0) {          /* fifty-fifty chance */
13         pay = 39;
14     } else {
15         pay = 34;
16     }
17
18     printf("$%d per hour\n", pay);
19 }
```

Code sinking

$$ac + bc = (a + b)c$$

I want french fries with my hamburger or coleslaw with my hamburger. I want french fries or coleslaw with my hamburger.

```
1     if (a == b) {
2         printf("equal");
3         c = d;
4     } else {
```



```
 5          printf("not equal");
 6          c = d;
 7      }

 8      if (a == b) {
 9          printf("equal");
10      } else {
11          printf("not equal");
12      }
13
14      c = d;
```

Code hoisting

$$ca + cb = c(a + b)$$

```
 1      if (a == b) {
 2          c = d;
 3          printf("equal");
 4      } else {
 5          c = d;
 6          printf("not equal");
 7      }

 8      c = d;
 9
10      if (a == b) {
11          printf("equal");
12      } else {
13          printf("not equal");
14      }
```

Could we still have hoisted the assignment statement if it had been **`a = b;`** instead of **`c = d;`**?

Never write an empty "else" section

Use an **`if-then-else`** statement only when there are two possible alternatives.

```
 1      if (a == b) {
 2          printf("equal");
 3      } else {
 4      }

 5      if (a == b) {
 6          printf("equal");
 7      }
```

Never write an empty "then" section

```
 1      if (a == b) {
 2      } else {
 3          printf("not equal");
 4      }

 5      if (a != b) {
 6          printf("not equal");
```



```
7     }
```

▼ **Homework 1.10: make the program simpler**

Hoist and sink if possible; do not write an empty ‘‘then’’ section or an empty ‘‘then’’ section.

```
1     if (a == b) {
2         printf("%d", c);
3         c = d;
4     } else {
5         c = d;
6     }

7     if (a == b) {
8         c = d;
9     } else {
10        printf("%d", c);
11        c = d;
12    }

13    if (d == 0) {
14        a = b + c;
15    } else {
16        a = b + c + d;
17    }

18    int a, b;
19
20    if (a == 2) {
21        b = b / 2;
22    } else if (a == 3) {
23        b = b / 3;
24    } else if (a == 4) {
25        b = b / 4;
26    } else if (a == 5) {
27        b = b / 5;
28    }
```

▲

Swap the “then” section and the “else” section

```
1     if (gore == bush) {
2         printf("They’re equal.\n");
3         printf("Please inform the House of Representatives\n");
4         printf("that the electoral college is hung.\n");
5     } else {
6         printf("There’s a clear winner.\n");
7         printf("Thank you, electors.\n");
8     }

9     if (gore != bush) {
10        printf("There’s a clear winner.\n");
11        printf("Thank you, electors.\n");
12    } else {
13        printf("They’re equal.\n");
```



```
14          printf("Please inform the House of Representatives\n");
15          printf("that the electoral college is hung.\n");
16      }
```

```
 1 /* before */
 2
 3 if (a == b) {
 4
 5
 6
 7
 8
 9 } else {
10
11 }
```

```
 1 /* after */
 2
 3 if (a != b) {
 4
 5 } else {
 6
 7
 8
 9
10
11 }
```

▼ **Homework 1.11: swap the "then" section and the "else" section**

```
 1      if (i <= 10) {
 2          printf("too few\n");
 3      } else {
 4          printf("too many\n");
 5      }

 6      if (temp < 72) {
 7          printf("too cold\n");
 8      } else {
 9          printf("too hot\n");
10      }
```

▲

Nested if 's

The ''then'' section can contain another **`if`**:

```
 1      if (profit <= loss) {
 2          printf("We're not making any money.\n");
 3      }
 4
 5      if (profit < loss) {
 6          printf("In fact, we're losing money.\n");
 7      }

 8      if (profit <= loss) {
 9          printf("We're not making any money.\n");
10          if (profit < loss) {
11              printf("In fact, we're losing money.\n");
12          }
13      }
```

The ''else'' section can contain another **`if`**:

```
14      if (profit > loss) {
15          printf("We're making money.\n");
16      } else {
```



```
17          printf("We're not making any money.\n");
18      }
19
20      if (profit < loss) {
21          printf("In fact, we're losing money.\n");
22      }

23      if (profit > loss) {
24          printf("We're making money.\n");
25      } else {
26          printf("We're not making any money.\n");
27          if (profit < loss) {
28              printf("In fact, we're losing money.\n");
29          }
30      }
```

A sequence of if-then-else's: K&R pp. 20−24, 57−58; King pp. 68−72

It's better to have the inner **if** in the ''else'' section rather than in the ''then'' section of the outer **if**. If necessary, swap the outer **if** as shown above. For example,

```
 1 /* before */
 2
 3 if (a != b) {
 4     if (a < b) {
 5         printf("less than");
 6     } else {
 7         printf("greater than");
 8     }
 9 } else {
10     printf("equal");
11 }
```

```
 1 /* during */
 2
 3 if (a == b) {
 4     printf("equal");
 5 } else {
 6     if (a < b) {
 7         printf("less than");
 8     } else {
 9         printf("greater than");
10     }
11 }
```

The following is the only exception to my rule to write every pair of curly braces. An **if** statement (with or without an ''else'' section) counts as a single statement (pp. 222−223). For example, lines 6−10 inclusive in **during** count as a single statement. Thus the ''else'' section beginning at line 5 in **during** contains only a single statement. Therefore the curly braces in **during** at the end of line 5 and in line 11 are optional. Remove them and line up the surviving pieces like this:

```
1       /* after */
2
3       if (a == b) {
4           printf("equal");
5       } else if (a < b) {
6           printf("less than");
7       } else {
8           printf("greater than");
9       }
```

Remove the curly braces around an ''else'' section only when they enclose a single **if** and nothing else, as shown above.

The above example is called a ''three-way'' **if**: it executes exactly one of three possible alternatives. There are even longer **if**'s:

```
10      if (b == c) {
11          printf("equal");
```

```
12      } else if (b < c) {
13          printf("less than");
14      } else if (b > c + 100) {
15          printf("much greater than");
16      } else {
17          printf("not all that much greater than");
18      }
```

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/leap1.c

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int year;
 6
 7     printf("Please type a year and press RETURN: ");
 8     scanf("%d", &year);
 9
10     if (year % 400 == 0) {
11         printf("%d is a leap year.\n", year); /* e.g., 1600, 2000 */
12     } else if (year % 100 == 0) {
13         printf("%d is not a leap year.\n", year); /* e.g., 1800, 1900 */
14     } else if (year % 4 == 0) {
15         printf("%d is a leap year.\n", year);
16     } else {
17         printf("%d is not a leap year.\n", year);
18     }
19 }
```

C, C++, Java, JavaScript, awk, C shell	**else if**
C and C++ preprocessors	**#elif**
Bourne, Korn, and Bourne-Again shells	**elif**
Perl	**elsif**
Tcl	**elseif**
troff, nroff	**.el .if**
m4	**.ifelse**

▼ Homework 1.12: unsnarl four badly written if's (not to be handed in)

Change each of the following four examples to a straight line of **if**'s as shown above. The answers are in this handout or the next.

```
1 if (guess >= n) {
2     if (guess > n) {
3         printf("You guessed too high.  Try again.\n");
4     }
5 } else {
6     printf("You guessed too low.  Try again.\n");
7 }
```

```
 8 if (year % 4 != 0) {
 9     if (year % 4 == 2) {
10         printf("%d: congressional election year\n", year);
11     } else {
12         printf("%d: local election year\n", year);
13     }
14 } else {
15     printf("%d: presidential election year\n", year);
16 }

17 if (a != b) {
18     if (a >= b) {
19         if (a > b + 100) {
20             printf("much greater than");
21         } else {
22             printf("not all that much greater than");
23         }
24     } else {
25         printf("less than");
26     }
27 } else {
28     printf("equal");
29 }
```

Here is the unsnarled version of the above **if**:

```
30 if (a == b) {
31     printf("equal");
32 } else if (a < b) {
33     printf("less than");
34 } else if (a > b + 100) {
35     printf("much greater than");
36 } else {
37     printf("not all that much greater than");
38 }
```

▲

The Metro North evactuation instructions

Listen for instructions from authorized personnel.

(1) Remain inside the train if possible. If not ...

(2) Go to next car through end doors. If unable ...

(3) Open side door and get out. If you can't ...

(4) Go out emergency windows.

```
1     if (possible) {
2         remain inside the train;
3     } else if (able) {
4         go to next car through end doors;
5     } else if (you can) {
6         open side door and get out;
7     } else {
8         go out emergency windows;
9     }
```



```
10      if (not possible) {
11          if (not able) {
12              if (you can't) {
13                  go out emergency windows;
14              } else {
15                  open side door and get out;
16              }
17          } else {
18              go to next car through end doors;
19          }
20      } else {
21          remain inside the train;
22      }
```

A guessing game

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <time.h>
 4
 5 main()
 6 {
 7     int n;
 8     int guess;
 9
10     srand(time(NULL));
11     n = rand();
12     printf("Guess what number I'm thinking of.\n");
13
14     do {
15         scanf("%d", &guess);
16         if (guess < n) {
17             printf("You guessed too low.  Try again.\n");
18         } else if (guess > n) {
19             printf("You guessed too high.  Try again.\n");
20         }
21     } while (guess != n);
22
23     printf("That's right!\n");
24 }
```

Remove unnecessary machinery by process of elimination

```
 1 if (a == b) {
 2     printf("equal");
 3 } else if (a < b) {
 4     printf("less than");
 5 } else if (a > b) {
 6     printf("greater than");
 7 }

 8 if (a == b) {
 9     printf("equal");
10 } else if (a != b) {
```



```
11     printf("not equal");
12 }

13 if (a == b) {
14     printf("equal");
15 }
16
17 if (a < b) {
18     printf("less than");
19 } else if (a > b) {
20     printf("greater than");
21 }
```

How not to use if

Rearrange the following program to eliminate the **`if`**, but keep the two **`printf`**'s. Do this whenever the body of a loop begins with an **`if`** that is true only during the first iteration, or the body ends with an **`if`** that is true only during the last iteration.

```
 1 /* Print the numbers from 1 to 10, one per line */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int i;
 7
 8     for (i = 1; i <= 10; i = i + 1) {
 9         if (i == 1) {
10             printf("Here are the numbers from 1 to 10.\n");
11         }
12         printf("%2d\n", i);
13     }
14 }

15 /* Print the numbers from 1 to 10, one per line. */
16 #include <stdio.h>
17
18 main()
19 {
20     int i;
21
22     for (i = 1; i <= 10; i = i + 1) {
23         printf("%2d\n", i);
24         if (i == 10) {
25             printf("That's all, folks!\n");
26         }
27     }
28 }
```

Spacing rules

See K&R p. 191, §A2.1 and King pp. 25−27 A group of one or more consecutive blanks, tabs, and/or newlines is called *whitespace.* There is one rule about where whitespace is prohibited, and one rule about where it is required. These rules are stated in terms of *tokens,* which are the words, numbers, quoted characters or strings, operators, or other punctuation marks that make up the source code of the program. The

tokens fall into two groups, alphanumeric and non-alphanumeric. Here are examples of both kinds.

alphanumeric tokens		*non-alphanumeric tokens*	
`main`	`10`	`+`	`[`
`int`	`010`	`-`	`]`
`void`	`0x10`	`.`	`?`
`const`	`10U`	`->`	`:`
`sizeof`	`10L`	`++`	`;`
`if`	`10UL`	`==`	`,`
`for`	`10.0`	`*=`	`{`
`typedef`	`10.0F`	`&=`	`}`
`extern`	`10.0L`	`&&`	`(`
`printf`	`10.0e5`	`<<`	`)`
`i`	`10.0e5F`	`<<=`	`'A'`
`my_func`	`10.0e5L`	`"a quoted string"`	

Here are the spacing rules.

(1) Whitespace is prohibited inside a token. Don't try to write

```
1 ma in
2 c out
3 < <
4 10 e 5
```

A comment delimiter is not a token. Even so, do not write

```
5 / /bad comment delimiter
6 / * bad opening comment delimiter */
7 /* bad closing comment delimiter * /
```

(2) Whitespace is required between two consecutive tokens that would otherwise be mistaken for a single token or a comment delimiter. There are three cases.

(2a) If the two tokens are alphanumeric, whitespace is always required between them. For example, whiespace is required between the **`int`** and the **`i`** in line 9. If we forget the whitespace, line 9 might still compile (thanks to line 8) but it would have a different meaning.

```
8 int inti = 10;
9 int i = 20;
```

(2b) If the two tokens are non-alphanumeric, whitespace is required between them only in the following exceptional

□

