Spring 2007 Handout 2

If-then-else vs. switch: K&R pp. 58–60, 223; King pp. 74–79

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/switch1.c

```
 1 /* Print the years 1992 to 2012, one per line, listing which ones are
 2 presidential, congressional, and local election years. */
 3 #include <stdio.h>
 4
 5 main()
 6 {
 7     int year;
 8
 9     for (year = 1992; year <= 2012; year = year + 1) {
10         printf("%d: ", year);
11
12         if (year % 4 == 0) {
13             printf("presidential election year\n");
14         } else if (year % 4 == 2) {
15             printf("congressional election year\n");
16         } else {
17             printf("local election year\n");
18         }
19     }
20 }
```

```
1992: presidential election year
1993: local election year
1994: congressional election year
1995: local election year
1996: presidential election year
1997: local election year
1998: congressional election year
1999: local election year
2000: presidential election year
2001: local election year
2002: congressional election year
2003: local election year
2004: presidential election year
2005: local election year
2006: congressional election year
2007: local election year
2008: presidential election year
```

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/switch2.c

All rights reserved ©2007 Mark Meretzky

```
 1 /* Same output as the above program, but use switch instead. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int year;
 7
 8     for (year = 1992; year <= 2012; year = year + 1) {
 9         printf("%d: ", year);
10
11         switch (year % 4) {      /* Always write the curly braces. */
12
13         case 0:
14             printf("presidential election year\n");
15             break;
16
17         case 2:
18             printf("congressional election year\n");
19             break;
20
21         default:
22             printf("local election year\n");
23             break;
24         }
25     }
26 }
```

The language does not the define the order in which the **`case`**’s are examined.

No breaks

—On the Web at
`http://i5.nyu.edu/~mm64/x52.9232/src/switch3.c`

```
 1 /* Same output as the above program, but use switch instead. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int year;
 7
 8     for (year = 1992; year <= 2012; year = year + 1) {
 9         printf("%d: ", year);
10
11         switch (year % 4) {
12
13         case 0:
14             printf("presidential/");
15
16         case 2:
17             printf("congressional/");
18
19         default:
20             printf("local\n");
21             break;
22         }
```

 All rights reserved ©2007 Mark Meretzky

```
23     }
24 }
```

```
1992: presidential/congressional/local
1993: local
1994: congressional/local
1995: local
1996: presidential/congressional/local
1997: local
1998: congressional/local
1999: local
2000: presidential/congressional/local
2001: local
2001: congressional/local
2003: local
2004: presidential/congressional/local
2005: local
2006: congressional/local
2007: local
2008: presidential/congressional/local
2009: local
2010: congressional/local
2011: local
2012: presidential/congressional/local
```

▼ Homework 2.1: write a switch statement without break's

Write a program named **partridge.c** that will print the words to the first five days of *A Partridge in a Pear Tree*. It will have exactly one variable, named **day**, one **for** loop, and one **switch** statement, with a **case** for each day of Christmas. Print an empty line after each day.

In what order must the **case**'s be written? Omit the **break** from each case except day 1; see K&R p. 59; King pp. 76–77. In case of malfunction, add a **default** after the last **case** with an error message printing the value of **day**. The words "partridge", "turtledoves", "hens", etc., must each appear exactly once in your program. Put the entire **switch** statement inside the **for** loop, which will count from 1 to 5. For simplicity, forget about the word "and" that comes after "turtledoves".

In a program that prints the first five days of Christmas, is it more natural to say **day<=5** or **day<6**? They are equally efficient.

```
On the 1 day of Christmas my true love gave to me
A partridge in a pear tree.

On the 2 day of Christmas my true love gave to me
Two turtledoves
A partridge in a pear tree.

On the 3 day of Christmas my true love gave to me
Three French hens
Two turtledoves
A partridge in a pear tree.          etc.
```

Then fix the word "and" with an **if-then-else** inside of one of the **case**'s. Also append the ordinal suffixes **st**, **nd**, **rd**, and **th** to the day numbers with another **switch** containing three **case**'s and one **default**. This **switch** must have four **break**'s. Your program must still have exactly one variable. It needs to print only the first five days of Christmas correctly. You get no credit if the words **On** or

 All rights reserved ©2007 Mark Meretzky

Christmas appear more than once in your program.

```
On the 1st day of Christmas my true love gave to me
A partridge in a pear tree.

On the 2nd day of Christmas my true love gave to me
Two turtledoves
And a partridge in a pear tree.

On the 3rd day of Christmas my true love gave to me
Three French hens
Two turtledoves
And a partridge in a pear tree.      etc.
```

Hand in only the last version of the program.

How many bytes in an int: K&R pp. 135, 204; King pp. 109–110

The parentheses are not needed around the name of a data type when it is the first word of a declaration:

```
int i = 10;      /* Don't need parentheses around the word "int". */
```

In all other cases, you must parenthesize the name of a data type. For example, the operand of **sizeof** must be parenthesized when it is the name of a data type such as **int**, but not when it is a variable such as **i**.

Use **%d** to print a number in decimal; use **%p** to print an address in whatever base is most appropriate for your machine.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int i = 10;
 6
 7     printf("The value of i is %d.\n", i);
 8     printf("The value of negative i is %d.\n", -i);
 9     printf("The address of i is %p.\n\n", &i);
10
11     printf("The size in bytes of i is %d.\n", sizeof i);
12     printf("The size in bytes of an int is %d.\n", sizeof(int));
13 }
```

```
The value of i is 10.
The value of negative i is -10.
The address of i is 7FFFB894.            May be different on your machine.

The size in bytes of i is 4.             May be different on your machine.
The size in bytes of an int is 4.        May be different on your machine.
```

Overflow: K&R p. 200 §A7, pp. 257–8; King pp. 109–111

A *bit* is a `0` or a `1`; it's short for *binary digit.* Four bits constitute one *nibble,* for example `1010`. Eight bits constitute one *byte,* for example `01000001`. There are 256 different bytes because there are 256 different eight-bit combinations of `1`'s and `0`'s, ranging from `00000000` to `11111111`.

 All rights reserved ©2007 Mark Meretzky

There are 65,536 different two-byte combinations of **`1`**'s and **`0`**'s, ranging from **`0000000000000000`** to **`1111111111111111`**. Therefore a two-byte **`int`** (or **`short`** or **`long`**) can hold any one of 65,536 different values, ranging from –32,768 to 32,767.

There are 4,294,967,296 different four-byte combinations of **`1`**'s and **`0`**'s, ranging from **`00000000000000000000000000000000`** to **`11111111111111111111111111111111`**. Therefore a four-byte **`int`** (or **`short`** or **`long`**) can hold any one of 4,294,967,296 different values, ranging from –2,147,483,648 to 2,147,483,647.

There are 18,446,744,073,709,551,616 different eight-byte combinations of **`1`**'s and **`0`**'s, ranging from **`0000000000000000000000000000000000000000000000000000000000000000`** to **`1111111111111111111111111111111111111111111111111111111111111111`**. Therefore an eight-byte **`int`** (or **`short`** or **`long`**) can hold any one of 18,446,744,073,709,551,616 different values, ranging from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

The following example assumes that **`sizeof(int)`** is 4. If your **`sizeof(int)`** is 2, change **`2147483647`** to **`32767`** and the expected output will be **`-32768`** instead of **`-2147483648`**.

```
1 #include <stdio.h>
2
3 main()
4 {
5     int i = 2147483647;    /* largest possible value for a 4-byte int */
6
7     i = i + 1;             /* Overflow causes no error message. */
8     printf("%d\n", i);
9 }
```

```
-2147483648
```

Don't write this loop if **`i`** is an **`int`** and **`sizeof(int)`** is 2:

```
1     /* delay loop */
2     for (i = 0; i <= 40000; i = i + 1) {
3     }
```

short, int, long: K&R p. 36, 196 King pp. 110–111

Don't bother to create **`short`**'s if you need only a few variables of this type; simply use **`int`**'s. Create **`short`**'s only when you need very many of them, i.e., an *array* of them.

The keyword **`int`** is optional in lines 1 and 3. Remove it.

On machines where **`sizeof(int) == sizeof(long)`**, line 7 might still work if accidentally wrote **`"d"`** instead of **`"ld"`**. But that's no reason to do it wrong.

```
 1     short int s = 32767;
 2     int i = 2147483647;
 3     long int tallsally = 2147483647;
 4
 5     printf("%d\n", s);
 6     printf("%d\n", i);
 7     printf("%ld\n", tallsally);     /* percent lowercase LD */
 8
 9     scanf("%hd", &s);
10     scanf("%d", &i);
11     scanf("%ld", &tallsally);       /* percent lowercase LD */
```

 All rights reserved ©2007 Mark Meretzky

char: K&R pp. 15–16, 36, 37, 195 King pp. 111–112, 116–118

The name **char** is a misnomer. A **char** is actually a one-byte integer:

```
 1      char c = 65;
 2      printf("%d\n", c);          /* It prints 65 */
```

One example of a number that you can store in a **char** is the ASCII code of a character. A **char** has room for the ASCII code of only one character. We'll talk about strings of characters after we do arrays (K&R pp. 22–24, 28–31; King pp. 139–140).

```
 3      char c = 65;                /* Puts 01000001 into c. */
 4
 5      printf("%d\n", c);          /* It prints 65 */
 6      printf("%c\n", c);          /* It prints A */
 7      putchar(c);                 /* Simpler way to do the same thing: it prints A */
 8
 9      /* Let the user type one character, and store its ASCII code into c: */
10      scanf("%c", c);
11      c = getchar();              /* A simpler way to do the same thing. */
```

Single quotes: K&R pp. 19–20, 37–38, 193–194; King p. 117

Enclose exactly one character in single quotes, even though K&R p. 193, §A2.5.2; permits more than one. The value of this expression is the ASCII code of the quoted character.

```
 1      char c = 65;                /* Puts 01000001 into c. */
 2      char d = 'A';               /* Puts 01000001 into d. */
 3
 4      printf("%d", c);            /* It prints 65 */
 5      printf("%c", c);            /* It prints A */
 6
 7      printf("%d", d);            /* It prints 65 */
 8      printf("%c", d);            /* It prints A */
```

Here is a trickier example:

```
 9      char c = ' ';               /* Put the byte 00100000 into c (32). */
10
11      printf("%c", c);            /* It prints a blank. */
12      printf("%d", c);            /* It prints 48 */
```

An even trickier example:

```
13      char c = '0';               /* Put the byte 00110000 into c (48). */
14      char d = 0;                 /* Put the byte 00000000 into d. */
15
16      printf("%c", c);            /* It prints 0 */
17      printf("%d", c);            /* It prints 48 */
18
19      printf("%c", d);            /* It prints nothing at all.  */
20      printf("%d", d);            /* It prints 0 */
```

To put the ASCII code of a non-printing character such as BEL into a variable,

```
21      /* All four of these statements create beep and put 00000111 into it. */
22      char beep = '\007';         /* octal */
23      char beep = '\x07';         /* hex */
24      char beep = 7;              /* decimal */
```

 All rights reserved ©2007 Mark Meretzky

```
25     char beep = '\a';          /* "alarm" */

26     char newline = '\n';
27     char tab = '\t';
28     char eos = '\0';           /* End of string: the byte 00000000 */
```

```
 1 /* Print a table of the ASCII codes of the printable characters. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     char c;
 7
 8     printf("decimal   octal     hex   ASCII\n");
 9
10     for (c = 32; c <= 126; c = c + 1) {       /* Iterates 95 times. */
11         printf("%7d %7o %7X %7c\n", c, c, c, c);
12     }
13 }
```

```
decimal   octal     hex   ASCII
     32      40      20
     33      41      21       !
     34      42      22       "
     35      43      23       #
     36      44      24       $
     37      45      25       %
     38      46      26       &
     39      47      27       '
     40      50      28       (        etc.
```

In the above program, replace the starting and ending numbers with single-quoted characters (K&R p. 249, §B2; King pp. 116–117):

```
for (c = ' '; c <= '~'; c = c + 1) {
```

Unicode characters

All the alphabets—English (i.e., Latin), Greek, Hebrew, Arabic, Japanese, Chinese, etc.—have been concatenated into one big alphabet containing 65,536 characters. Each character therefore occupies 16 bits. See the charts at `http://www.unicode.org`

Don't use single quotes for a Unicode character. Write them in as four-digit hexadecimal numbers:

```
1 #include <stddef.h>  /* because wchar_t is not a keyword like char, K&R p. 193 */
2
3     wchar_t greek_alpha = 0x03B1;   /* lowercase alpha: decimal 945 */
4     wchar_t hebrew_alef = 0x05D0;   /* decimal 1488 */
5     wchar_t arabic_alef = 0x0627;   /* decimal 1575 */
```

unsigned short, unsigned, unsigned long: K&R pp. 36, 196; King pp. 110–113

`unsigned short`, `unsigned int`, and `unsigned long` variables cannot hold negative numbers. An `unsigned short` is the same size as a `short`, an `unsigned int` is the same size as an `int`, and an `unsigned long` is the same size as a `long`.

 All rights reserved ©2007 Mark Meretzky

A two-byte **int** (or **short** or **long**) can hold any one of 65,536 different values, ranging from –32,768 to 32,767. A two-byte **unsigned int** (or **unsigned short** or **unsigned long**) can also hold any one of 65,536 different values, but they range from 0 to 65,535. In both cases there are 65,536 different values because there are 65,536 different two-byte combinations of **1**’s and **0**’s, ranging from **0000000000000000** to **1111111111111111**.

A four-byte **int** (or **short** or **long**) can hold any one of 4,294,967,296 different values, ranging from –2,147,483,648 to 2,147,483,647. A four-byte **unsigned int** (or **unsigned short** or **unsigned long**) can also hold any one of 4,294,967,296 different values, but they range from 0 to 4,294,967,295. In both cases there are 4,294,967,296 different values because there are 4,294,967,296 different four-byte combinations of **1**’s and **0**’s, ranging from **00000000000000000000000000000000** to **11111111111111111111111111111111**.

An eight-byte **int** (or **short** or **long**) can hold any one of 18,446,744,073,709,551,616 different values, ranging from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. An eight-byte **unsigned int** (or **unsigned short** or **unsigned long**) can also hold any one of 18,446,744,073,709,551,616 different values, but they range from 0 to 18,446,744,073,709,551,615. In both cases there are 18,446,744,073,709,551,616 different values because there are 18,446,744,073,709,551,616 different eight-byte combinations of **1**’s and **0**’s, ranging from **0000000000000000000000000000000000000000000000000000000000000000** to **1111111111111111111111111111111111111111111111111111111111111111**.

The keyword **int** is optional in lines 1,2, and 3. Remove it.

```
 1      unsigned short int us = 65535;
 2      unsigned int u = 4294967295;
 3      unsigned long int ul = 4294967295;
 4
 5      printf("%u\n", us);
 6      printf("%u\n", u);
 7      printf("%lu\n", ul);     /* percent lowercase LU */
 8
 9      scanf("%hu", &us);
10      scanf("%u", &u);
11      scanf("%lu", &ul);           /* percent lowercase LU */
```

float, double, and long double: K&R pp. 9, 36, 37, 196, 257–258; King pp. 114–116

```
 1      float f = 3.14159;
 2      double d = 3.141592654;
 3      long double daysjourney = 3.141592654;
 4
 5      f = 6.0225e23;               /* Avogadro’s number: 6.0225×10^23 */
 6      d = 6.0225e23;
 7      daysjourney = 6.0225e23;
 8
 9      printf("%f\n", f);   /* or %e or %g */
10      printf("%f\n", d);
11      printf("%Lf\n", daysjourney);/* L must be uppercase */
12
13      scanf("%f", &f);
14      scanf("%lf", &d);            /* percent lowercase LF */
15      scanf("%Lf", &daysjourney);  /* L must be uppercase */
```

```
 1 #include <stdio.h>
 2
 3 main()
```

 All rights reserved ©2007 Mark Meretzky

```
 4 {
 5     double d;
 6
 7     scanf("%lf", &d);             /* percent lowercase LF */
 8     printf("%10.5f\n", d);        /* percent ten point five lowercase F */
 9 }
```

`1.234567`	*I typed this.*
`   1.23457`	*It printed three blanks before the number, and rounded to 5 decimal places.*

Is the number rounded or truncated to 5 decimal places? K&R p. 13, p. 153 bullet 4, and p. 244 bullet 2 don't say. Use `%.2f` to print a `float` or `double` containing an amount of money.

Conversion characters for scanf (K&R pp. 157–159, 245–246; King pp. 494–496) and printf (K&R pp. 153–155, 243–244; King pp. 488–490)

<table>
<tr><th></th><th>scanf</th><th>printf</th></tr>
<tr><td>char</td><td>%c</td><td rowspan="3">%c %d %o %x %X</td></tr>
<tr><td>short</td><td>%hd %ho %hx</td></tr>
<tr><td>int</td><td>%d %o %x</td></tr>
<tr><td>long</td><td>%ld %lo %lx</td><td>%ld %lo %lx %lX</td></tr>
<tr><td>unsigned short</td><td>%hu %ho %hx</td><td rowspan="2">%c %u %o %x %X</td></tr>
<tr><td>unsigned</td><td>%u %o %x</td></tr>
<tr><td>unsigned long</td><td>%lu %lo %lx</td><td>%lu %lo %lx %lX</td></tr>
<tr><td>float</td><td>%e %f %g</td><td rowspan="2">%e %f %g</td></tr>
<tr><td>double</td><td>%le %lf %lg</td></tr>
<tr><td>long double</td><td>%Le %Lf %Lg</td><td>%Le %Lf %Lg</td></tr>
<tr><td>pointer</td><td>%p</td><td>%p</td></tr>
</table>

▼ Homework 2.2: print the sizeof each data type

Write a program to print the size in bytes of the eight data types **`char`**, **`wchar_t`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, and **`long double`**. Some versions of C have no **`long double`**. Verify that an **`unsigned`** is the same size as an **`int`**.

An array of integers: K&R pp. 22, 86, 219; King pp. 139–143

```
 1 /* Print the number of days in a non-leap year. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int sum = 0;
 7
 8     sum = sum + 31;  /* January */
 9     sum = sum + 28;  /* Feruary */
10     sum = sum + 31;  /* March */
11     sum = sum + 30;  /* April */
12     sum = sum + 31;  /* May */
13     sum = sum + 30;  /* June */
```

 All rights reserved ©2007 Mark Meretzky

```
14      sum = sum + 31;  /* July */
15      sum = sum + 31;  /* August */
16      sum = sum + 30;  /* September */
17      sum = sum + 31;  /* October */
18      sum = sum + 30;  /* November */
19      sum = sum + 31;  /* December */
20
21      printf("%d\n", sum);
22 }
```

The subscript of an array is always enclosed by **[**square brackets**]**, not parentheses. If you create an array containing 12 numbers as in lines 6–19 below, the subscripts go from 0 to 11. *There is no subscript 12!*

If you provide an initial value for each array element, as we did in lines 7–18, then you don't have to write the **12** in line 6 (but you still have to write the square brackets in line 6).

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/array1.c

```
 1 /* Print the number of days in a non-leap year. */
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6      int a[12] = {
 7           31,      /* January */
 8           28,      /* February */
 9           31,      /* March */
10           30,      /* April */
11           31,      /* May */
12           30,      /* June */
13           31,      /* July */
14           31,      /* August */
15           30,      /* September */
16           31,      /* October */
17           30,      /* November */
18           31,      /* December */       /* Trailing comma optional: KR p. 218-219. */
19      };
20
21      int i;
22      int sum = 0;
23
24      for (i = 0; i < 12; i = i + 1) {
25           sum = sum + a[i];
26      }
27
28      printf("%d\n", sum);
29 }
```

```
365
```

In the above program, the subscripts of the months ranged from 0 to 11 inclusive. In the following program, they range from 1 to 12 inclusive. In line 6, change **13** to **12 + 1**: always write in terms of the magic number.

```
 1 /* Print the number of days in a non-leap year. */
```

 All rights reserved ©2007 Mark Meretzky

```
 2 #include <stdio.h>
 3
 4 main()
 5 {
 6     int a[13] = {
 7         0,       /* dummy element to make January have subscript 1 */
 8         31,      /* January */
 9         28,      /* February */
10         31,      /* March */
11         30,      /* April */
12         31,      /* May */
13         30,      /* June */
14         31,      /* July */
15         31,      /* August */
16         30,      /* September */
17         31,      /* October */
18         30,      /* November */
19         31       /* December */
20     };
21
22     int i;
23     int sum = 0;
24
25     for (i = 1; i <= 12; i = i + 1) {
26         sum = sum + a[i];
27     }
28
29     printf("%d\n", sum);
30 }
```

Use the above array (dummy element, subscripts ranging from 1 to 12 inclusive) in the following three homeworks.

Use an array to telescope a program

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int dependents;
 6
 7     printf("How many dependents do you have? ");
 8     scanf("%d", &dependents);
 9
10     if (dependents < 0) {
11         printf("Must be a non-negative number.\n");
12     } else if (dependents == 0) {
13         printf("You can deduct 30 dollars.\n");
14     } else if (dependents == 1) {
15         printf("You can deduct 50 dollars.\n");
16     } else if (dependents == 2) {
17         printf("You can deduct 65 dollars.\n");
18     } else if (dependents == 3) {
19         printf("You can deduct 95 dollars.\n");
20     } else if (dependents == 4) {
```

 All rights reserved ©2007 Mark Meretzky

```
21          printf("You can deduct 105 dollars.\n");
22      } else if (dependents == 5) {
23          printf("You can deduct 120 dollars.\n");
24      } else {
25          printf("You can deduct 124 dollars.\n");
26      }
27 }
```

```
28 #include <stdio.h>
29
30 main()
31 {
32     int deduction[] = {
33         30,             /* 0 dependents */
34         50,             /* 1 dependent  */
35         65,             /* 2 dependents */
36         95,             /* 3 dependents */
37         105,            /* 4 dependents */
38         120             /* 5 dependents */
39     };
40
41     int dependents;
42
43     printf("How many dependents do you have? ");
44     scanf("%d", &dependents);
45
46     if (dependents < 0) {
47         printf("Must be a non-negative number.\n");
48     } else if (dependents > 5) {
49         printf("You can deduct 124 dollars.\n");
50     } else {
51         printf("You can deduct %d dollars.\n", deduction[dependents]);
52     }
53 }
```

Swap the values of two variables

```
/* bad */                            /* good */
i = j;                               temp = i;
j = i;                               i = j;
                                     j = temp;
```

Bubble sort the numbers in an array

Would it be merely an inefficiency or a full-scale disaster to change the **>** to **>=** in line 30?

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

```
 1 /* Bubble sort an array of 10 ints into ascending order.  The for loop in lines
 2 29-38 will move the array elements part of the way into the correct order.
 3
 4 If some moves were made, it means that we should execute this for loop again to
 5 see if additional moves will be made.  In this case, flag is set to 1 to make
 6 the do-while loop execute the for loop again.
 7
```

 All rights reserved ©2007 Mark Meretzky

```
 8 If no moves were made, it means that the elements are already in order.  In this
 9 case, flag remains 0 and the do-while loop terminates. */
10
11 #include <stdio.h>
12
13 main()
14 {
15     int a[10];
16     int i;       /* index into the array */
17     int flag;    /* set to 1 to ensure one more trip */
18     int temp;    /* temporary storage for exchanging values */
19
20     /* Initialize the array with the numbers to be sorted. */
21     printf("Type %d numbers.  Press RETURN after each one.\n", 10);
22     for (i = 0; i < 10; ++i) {
23         scanf("%d", &a[i]);
24     }
25
26     /* Bubble sort the array into ascending order. */
27     do {
28         flag = 0;
29         for (i = 0; i < 9; ++i) {
30             if (a[i] > a[i+1]) {
31                 temp = a[i];     /* swap a[i] and a[i+1] */
32                 a[i] = a[i+1];
33                 a[i+1] = temp;
34
35                 flag = 1;
36             }
37             printf("debug: i == %d\n", i);
38         }
39     } while (flag == 1);
40
41     /* Output the array. */
42     for (i = 0; i < 10; ++i) {
43         printf("%11d\n", a[i]);
44     }
45 }
```

Last hired, first fired

A *stack* is a classic information storage and retrieval system. Accountants call it a ‘‘lifo’’ list: last in, first out. See K&R p. 77; King p. 187.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 main()
 5 {
 6     int val[100];    /* the stack itself.  Your machine may need "long". */
 7     int sp = 0;      /* stack pointer */
 8
 9     int ss;          /* Your machine may need "long". */
10
```

 All rights reserved ©2007 Mark Meretzky

```
11      printf("To hire a person, type their social security number.\n");
12      printf("To fire the most recently hired person, type a zero.\n");
13      printf("To quit, type a negative number.\n");
14
15      for (;;) {
16          scanf("%d", &ss);
17          if (ss < 0) {
18              break;                          /* K&R pp. 64-65; King pp. 97-98 */
19          }
20
21          if (ss > 0) {                       /* hire */
22              if (sp < 100) {
23                  val[sp] = ss;
24                  sp++;
25              } else {
26                  printf("Sorry, there are already %d employees.\n", 100);
27              }
28          } else {                            /* fire */
29              if (sp > 0) {
30                  --sp;
31                  printf("Firing number %d\n", val[sp]);
32              } else {
33                  printf("Sorry, there's no one left to fire.\n");
34              }
35          }
36      }
37  }
```

```
To hire a person, type their social security number.
To fire the most recently hired person, type a zero.
To quit, type a negative number.
10                          You type the numbers in italics.
20
30
0
Firing number 30.
0
Firing number 20.
40
0
Firing number 40.
0
Firing number 10.
−1
```

▼ Homework 2.3: is today the last day of the month?

Write a program named **monthend.c** that will input two numbers representing a month and day. It will then print one line telling whether or not the date is the last of the month.

Begin by **printf**'ing one line of instructions for the user. Then call **scanf** to input the two **int**'s. Do this with two individual **scanf**'s or one big **scanf**:

```
scanf("%d%d", &month, &day);
```

Then write an **if** statement that will execute one of two possible **printf**'s.

 All rights reserved ©2007 Mark Meretzky

You get credit only if the program contains exactly three **printf**'s and one **if-then-else**. Use exactly two variables, which must be **int**'s. Name them **month** and **day**.

Ignore leap years. You get no credit if you handle leap years, even if you do it correctly. Do not write twelve **if**'s. Do not use **&&** or **||**. Keep the array of month sizes shown above: do not change the numbers to 31, 59, 90, 120, etc.

In the following example, the user types the numbers in *italics*.

```
Input the month number (1 - 12) and day.  Press RETURN after each number.
2
22
It's not the last day of the month.
```

The numbers 31, 28, 31, 30, etc. must appear only in the array, not in any **if**. Don't program like this:

```
 1      if (month == 1) {             /* January */
 2          if (day == 31) {
 3              printf("It's the last day of the month.\n");
 4          } else {
 5              printf("It's not the last day of the month.\n");
 6          }
 7      } else if (month == 2) {      /* February */
 8          if (day == 28) {
 9              printf("It's the last day of the month.\n");
10          } else {
11              printf("It's not the last day of the month.\n");
12          }
13      } else if (month == 3) {      /* March */
14          if (day == 31) {
15              printf("It's the last day of the month.\n");
16          } else {
17              printf("It's not the last day of the month.\n");
18          }
19      } else if (month == 4) {      /* April */
20          etc.
```

or like this:

```
21      if (month == 1 && day == 31 ||
22          month == 2 && day == 28 ||
23          month == 3 && day == 31 ||
24          month == 4 && day == 30 || etc.) {
25          printf("It's the last day of the month.\n");
26      } else {
27          printf("It's not the last day of the month.\n");
28      }
```

▼ Homework 2.4: what is tomorrow's date?

Write a program named **tomorrow.c** that will input three numbers representing a month, day, and year. It will then print three numbers representing the date that is immediately after the input date.

Begin by **printf**'ing one line of instructions for the user. Then call **scanf** to input the three **int**'s.

You get credit only if the program contains exactly two **printf**'s and two **if-then-else**'s. (In other words, the words **printf**, **if**, and **else** must each appear exactly twice). Use exactly three

 All rights reserved ©2007 Mark Meretzky

variables, which must be **int**'s. Name them **month**, **day**, and **year**. You get no credit if you write **day=1** more than once.

Ignore leap years. You get no credit if you handle leap years, even if you do it correctly. Do not write twelve **if**'s. Do not use **&&** or **||**. Keep the array of month sizes shown above: do not change the numbers to 31, 59, 90, 120, etc. Do not convert to Julian (K&R p. 111) and convert back to month/day/year. Do not use pointers. Hand in your program after it works correctly for the eight dates in column 1 of the next Homework.

In the following example, the user types the numbers in *italics.*

```
Input the month, day, and year.  Press RETURN after each number.
2
22
2007
Tomorrow is 2-23-2007.
```

Do the two examples below produce the same output? If so, is one of them better than the other? Follow this principle in this homework and the next.

```
 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      }
```

▲

▼ Homework 2.5: when is the baby due?

Write a program named **due.c** that will input three numbers representing a month, day, and year. It will then print three numbers representing the date that is 280 days after the input date. In the following example, the user types the numbers in *italics.*

```
Input the month, day, and year.  Press RETURN after each number.
1
1
2007
The baby is due on 10-8-2007.
```

Hand in your program after it works correctly for the following twelve dates:

 All rights reserved ©2007 Mark Meretzky

Start	*End*	
`12 30 2007`	`10  6 2008`	*boundary tests around Dec. 31*
`12 31 2007`	`10  7 2008`	
` 1  1 2008`	`10  8 2008`	
` 3 25 2008`	`12 30 2008`	
` 3 26 2008`	`12 31 2008`	
` 3 27 2008`	` 1  1 2009`	
` 2 27 2008`	`12  4 2008`	*boundary tests around Feb. 28*
` 2 28 2008`	`12  5 2008`	
` 3  1 2008`	`12  6 2008`	
` 5 23 2008`	` 2 27 2009`	
` 5 24 2008`	` 2 28 2009`	
` 5 25 2008`	` 3  1 2009`	

Simply wrap a **for** loop around the **if** statements of the previous homework. The **for** loop must iterate 280 times. During each iteration, advance one day. After the loop is over, print the answer.

You get credit only if the program contains exactly two **printf**'s, one **for**, and two **if-then-else**'s. Use exactly four variables, which must be **int**'s. Name them **month**, **day**, **year**, and **i**. You get no credit if you write **day=1** more than once.

Ignore leap years. You get no credit if you handle leap years, even if you do it correctly. Do not write twelve **if**'s. Do not use **&&** or **||** Keep the array of month sizes shown above: do not change the numbers to 31, 59, 90, 120, etc. Do not convert to Julian (K&R p. 111), add 280, subtract 365 if necessary, and convert back from Julian to month/day/year. Do not use pointers.

✎ Extra credit. Write a faster version that loops one month at a time instead of one day at a time. You get extra credit only if your program contains exactly two **printf**'s, one **while**, and one **if-then-else**. Use exactly three variables, which must be **int**'s. Name them **month**, **day**, **year**. Do not attempt this problem without first doing the non-extra credit part.
▲

Evaluate an expression: K&R pp. 52–54; King pp. 54–59

We've seen expressions such as **a+b** in several places: to the right of the **=** in an assignment statement, as an argument to **printf**, etc.

```
i = a + b;
printf("%d\n", a + b);
```

When the computer computes the value of an expression, we say that the computer *evaluates* the expression.

The simplest expressions consist of a variable or a literal; see K&R p. 200, §A7.2; King p. 45. It takes little work to evaluate such an expression. For example,

```
i = a;                        /* The expression "a" is a variable. */
printf("%d\n", a);            /* The expression "a" is a variable. */

i = 10;                       /* The expression "10" is a literal. */
printf("%d\n", 10);           /* The expression "10" is a literal. */
```

Binary operators

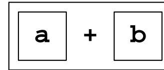
Two expressions can be joined with a *binary operator* such as **+ - * /** to make a bigger expression. All the binary operators are listed in the table on K&R p. 53; King p. 595. Except for lines 2 and 13 (the

 All rights reserved ©2007 Mark Meretzky

unary and ternary lines), everything listed there is a binary operator. We call them *binary* because they join two expressions, not because they have something to do with binary vs. decimal.

For example, **a** and **b** are expressions; therefore **a+b** is an expression. We say that **a** and **b** are *subexpressions* of **a+b**. **a** is the *left operand* of the **+** and **b** is the *right operand* of the **+**.

When evaluating **a+b**, the computer first evaluates the subexpressions **a** and **b**, and then it executes the addition to evaluate the whole expression **a+b**. We draw three boxes to show the order in which the computer evaluates these three expressions: innermost boxes first, outermost box last.



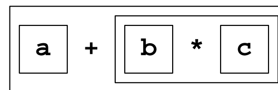
Order of evaluation

Of the three expressions in the diagram above, the big expression **a+b** must certainly be evaluated last. But which expression is evaluated first, **a** or **b**? The answer will be different in each vendor's version of C.

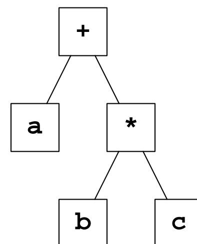
If one box is enclosed in another (as **a** is enclosed in **a+b**), then the inner box is always evaluated before the outer box. But if two boxes do not enclose one another (like **a** and **b**), then it is impossible to predict which of the two will be evaluated first. Don't worry: in the case of **a+b**, it doesn't matter. But see the last paragraph on K&R p. 52, and the second paragraph of K&R p. 200, §A7.

Operator precedence: King pp. 47–48

An expression can have more than one binary operator, for example **a+b*c**. Which of the following is the correct diagram for that expression? Does the computer execute the multiplication first or the addition first?



/* multiplication before addition */



The table on K&R p. 53 (ignoring lines 2 and 13), King p. 595 says that * has higher precedence than **+**. Therefore the computer executes the multiplication before the addition. Stated more technically, **b*c** is a subexpression of **a+b*c**, but **a+b** is not. **b*c** is the right operand of the **+**.

To evaluate an expression, the computer always evaluates the smallest subexpressions first—the individual variables and constants. Then it evaluates the next-to-smallest subexpressions. It keeps stepping back to take in a wider and wider view until it sees the Big Picture—the value of the entire expression.

Warning: most binary operators are a single character, but some are two or three characters:

```
a + b                    a -> b
a - b                    a >> b
a > b                    a >>= b
```

You can't have two binary operators with no tokens in between, so **->** cannot be the operator **-** followed by the operator **>**.

 All rights reserved ©2007 Mark Meretzky

▼ Homework 2.6: draw the boxes

Draw the boxes to show the order of evaluation in each of the following nine expressions. Use the table on K&R p. 53; King p. 595, ignoring lines 2 and 13.

```
a * b + c                               a >>= b >= c

a *= b + c                              a -> b > c -> d

a - b > c                               a || b | c

a -> b > c                              a == b && c == d && e == f


a *= b && c ^ d == e << f * g -> h + i > j & k | l || m , n
```

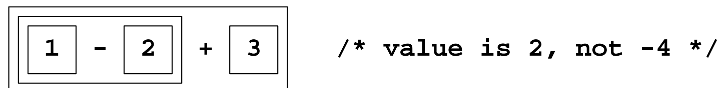
▼ Homework 2.7: xerox K&R p. 53; King p. 595 (not to be handed in)

Hang up a xerox of Table 2–1 on K&R p. 53; King p. 595. Label the second line *unary;* that's the line that contains **`++`**. Label the thirteenth line *ternary;* that's the line that contains **`?:`**. Circle the three exceptional "right to left" associativities.

Associativity

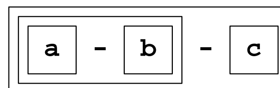
```
a - b + c
1 - 2 + 3
```

What happens if an expression contains two or more binary operators at the same level in the table on K&R p. 53; King p. 595, all competing to sink their teeth into the same subexpression? For example, both of the binary operators in the expression **`1-2+3`** are on line 4, and they both want to attack the **`2`**. Is the value of the whole expression 2 or –4?



The Associativity column of the table on K&R p. 53; King p. 595 says that whenever there are two or more consecutive binary operators on line 4, they are evaluated from left to right. Therefore the subtraction in the above example is executed before the addition, and the value of the whole expression is 2. Stated more technically, **`1-2`** is a subexpression of **`1-2+3`**, but **`2+3`** is not.

Another example is



▼ Homework 2.8: draw the boxes

Draw the boxes to show the order of evaluation in each of the following eight expressions. Use the table on K&R p. 53 including the associativity column if necessary, ignoring lines 2 and 13; King p. 595.

All rights reserved ©2007 Mark Meretzky

```
a + b - c                         a -> b . c

a * b / c                         a -> b -> c

a / b * c                         a = b % c

a == b != c                       d = c = b = a
```

An expression whose subexpressions are evaluated in an unpredictable order

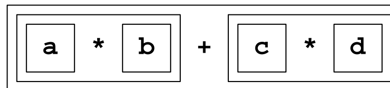
The associativity of the operators comes into play only when there are two or more consecutive operators with the same precedence. *Consecutive* means that the operators are all competing for the same operand. For example, both operators are competing to sink their teeth into the **b**:

```
a * b * c
```

or

```
a * b / c
```

The associativity is ignored when the operators with the same precedence are not consecutive, for example the ***** and ***** in the expression



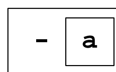
In this expression we know that the **+** is executed last, but it is impossible to predict whether the first ***** or the second ***** will be executed first. Stated more technically, it is impossible to predict which of the two subexpressions, **a*b** or **c*d**, will be evaluated first.

Unary operators

An expression can be augmented with a *unary operator* such as **- & sizeof** to make a bigger expression. All the unary operators are listed on line 2 of the table on K&R p. 53; King p. 595. Everything on that line is a unary operator. We call them *unary* because they each take only one expression, unlike the binary operators.

For example, **a** is an expression; therefore **-a** is also an expression. We say that **a** is a subexpression of **-a**, and that **a** is the operand of the **-**.

When evaluating **-a**, the computer first evaluates the subexpression **a**, and then it executes the negation to evaluate the whole expression **-a**. We draw two boxes to show the order in which the computer evaluates these two expressions.



How to tell the binary and unary operators apart

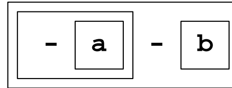
The four tokens **+ - *** and **&** can represent either a binary or a unary operator. They represent a binary operator when there is an expression on both sides of them. Otherwise they represent a unary operator.

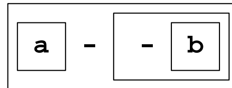
```
a - b          /* binary */
-a             /* unary */
```

 All rights reserved ©2007 Mark Meretzky

```
-a - b        /* first dash is unary, second dash is binary */
a - -b        /* first dash is binary, second dash is unary. */
```

Unary **-** has higher precedence than binary **-**



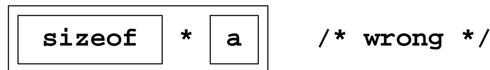


There must be white space between the two dashes in the last example, because a pair of dashes with no white space between is always the **--** unary operator. Ditto for **++** and **&&**. See K&R p. 192, §A2.1; King p. 25.

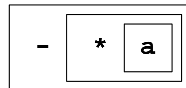
Warning: most unary operators are a single character, but some are two or more characters:

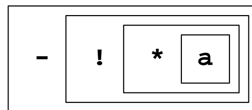
```
-a                                  --a
+a                                  ++a
&a                                  sizeof a
*a                                  (int)a
!a                                  (unsigned long)a
```

The word **sizeof** is not an expression—it's just an operator.



Put no white space between a unary operator and its operand unless required by Handout 1, p. 35, rule (2). You can't have two binary operators with no operands in between, but you can have two (or more) unary operators with no operands in between:





▼ Homework 2.9: draw the boxes

Draw the boxes to show the order of evaluation in each of the following twelve expressions. Use the table on K&R p. 53, ignoring line 13 (the ternary line); King p. 595.

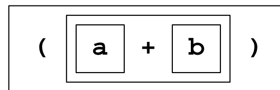
 All rights reserved ©2007 Mark Meretzky

```
a & &b                        *++a

a - &b                        -*++a

a & -b                        *a = *b

a * ++b                       *++a = *++b

sizeof *a                     *a *= *b

a < -1 || a > 10              a = !!b
```

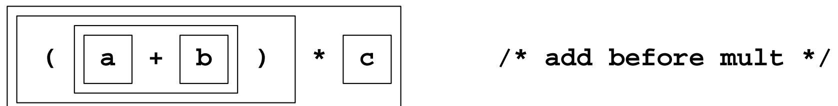
Parentheses: K&R pp. 17, 200–201 §A7.2; King pp. 54–56

Parentheses serve three totally different purposes in C expressions. A pair of parentheses is an operator on line 1 of the table on K&R p. 53; King p. 595. A pair of parentheses is also part of an operator on line 2. What follows is the third purpose of parentheses.

An expression can be surrounded by parentheses to make an even bigger expression. For example, **a+b** is an expression; therefore **(a+b)** is also an expression. The value of **(a+b)** is the same as that of **a+b**.



As in algebra and all computer languages, parentheses override the default precedence and associativity of the operators, i.e., the default nesting of the boxes.



▼ Homework 2.10: remove redundant parentheses

Remove the redundant parentheses from the following ten expressions, but keep the parentheses that might change the value of the expression.

```
(a * b) + (c * d)             (a % 8) == 7
(a == b) && (c == d)          (a) / (2)
(a & b) == (c & d)            (a->b).c
a = (b == c)                  ++(*p)
(a = b) == c                  *(++p)
```

Memorize at least this much of the table on K&R p. 53; King p. 595:

1 *the unary operators:* **-** *(for negation, not subtraction), etc.*
2 *** / %**
3 **+ -**
4 *the comparison operators:* **==**, *etc.*
5 **&&**

 All rights reserved ©2007 Mark Meretzky

```
6 ||
7 =
```

▲

An operator with a side effect: K&R p. 53; King pp. 50–52

In the statement

```
a = b + c;
```

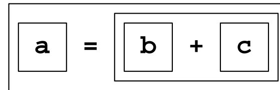
what causes the **+** to be executed before the **=**? The table on K&R p. 53; King p. 595 shows that the familiar **=** is actually a binary operator. Therefore the

```
a = b + c
```

in the statement

```
a = b + c;
```

is actually an expression. The **+** is executed before the **=** because it is higher than **=** in the table on K&R p. 53; King p. 595.



A *statement* in C is usually just an expression with a semicolon after it (K&R p. 55; King p. 57). To *execute* a statement is merely to evaluate the expression of which it consists.

The binary operator **=** changes the value of its left operand. This operand can therefore be only a variable, not a constant or a more complicated expression.

```
a = 10;         /* legal: left operand of = is a variable */
10 = a;         /* illegal: left operand of = is a constant */
(a + b) = 10;   /* illegal: left operand of = is a more complicated expression */
```

An expression that could legally be the left operand of **=** is called an *lvalue;* see K&R p. 197, §A5; King pp. 51–52. The "l" stands for "left". The only example of an lvalue that we have discussed so far is an expression that consists of one variable. The left operand of the other binary operators **+ - * / %** does not have to be an lvalue.

We say that the **=** operator has a *side effect* because it changes the value of a variable. When a normal expression such as **a+b** is evaluated, only one thing happens: the computer computes the value of the expression **a+b**, which may be used as part of a bigger expression. But when an expression such as **a=b** is evaluated, two things happen: **a** gets a new value, and the computer computes the value of the expression **a=b**, which may be used as part of a bigger expression. The value of the expression **a=b** is the new value of **a**.

```
 1 #include <stdio.h>
 2
 3 main()
 4 {
 5     int a = 1;
 6     int b = 2;
 7
 8     printf("%d\n", a + b);        /* value of a does not change */
 9     printf("%d\n", a = b);        /* value of a changes */
10     printf("%d\n", a);
11 }
```

 All rights reserved ©2007 Mark Meretzky

```
3
2
2
```

```
    if (a == b) {      /* good */
    if (a = b) {       /* bad, but no error message */
```

You can change each example on the left to the one on the right.

```
 1 a = newval;
 2 printf("%d\n", a);                    printf("%d\n", a = newval);

 3 a = newval;
 4 b = a;                                b = a = newval;

 5 a = newval;
 6 b = a;
 7 c = b;                                c = b = a = newval;

 8 a = newval;
 9 c[a] = b;                             c[a = newval] = b;

10 a = newval;                           /* Need parens to execute = before == */
11 if (a == b) {                         if ((a = newval) == b) {

12 a = newval;                           /* Code sinking: */
13 while (a == b) {                      while ((a = newval) == b) {
14     blah blah blah;                       blah blah blah;
15     a = newval;                       }
16 }

17 c = getchar();                        /* K&R pp. 15-17; King pp. 121-122 */
18 while (c != EOF)) {                   while ((c = getchar()) != EOF) {
19     blah blah blah;                       blah blah blah;
20     c = getchar();                    }
21 }
```

▼ **Homework 2.11: required reading: K&R pp. 15–24**

Read all the programs whose main loop is the **`while-getchar`** shown above.

Assignment operators: K&R pp. 50–51, 208–209; King pp. 50–52

The same expression often appears on either side of an **=** operator:

```
1 a = a + b;                             a += b;
2 a = a - b;                             a -= b;
3 a = a * 10;                            a *= 10;
4 a[b] = a[b] / c;                       a[b] /= c;
5 a[b+c+d] = a[b+c+d] / e;               a[b+c+d] /= e;
```

The **+=** operator has a side effect: it gives a new value to its left operand, which must be an lvalue. The new value is the sum of the value of the right operand and the old value of the left operand. The value of the expression **a+=b** is the new value of the left operand of the **+=**, just as the value of the expression **a=b** is the new value of the left operand of the **=**.

 All rights reserved ©2007 Mark Meretzky

Don't bother to change **a=a+1** to **a+=1** because there is a better abbreviation below: **++a**. Never write either of these expressions.

▼ Homework 2.12: rewrite using assignment operators

Where possible, rewrite the following expressions using assignment operators other than **=**. The expression ***a** is an lvalue. Addition is commutative, subtraction isn't.

```
a = a / 10                              a = b + a
a = a % 10                              a = b - a
a = a & b                               a = a + a
a = a >> 10                             a = a * 2
*a = *a + b                             a = a -> b
a = b - (c - a)
```

A unary operator with a side effect: K&R pp. 18, 46–48, 203; King pp. 53–54

The unary operator **++** has a side effect: it adds 1 to the value of its operand, which must be an lvalue. For example, when the expression **++a** is evaluated, two things happen: **a** gets a new value, and the computer computes the value of the expression **++a**, which may be used as part of a bigger expression. The value of the expression **++a** is the new value of **a**, just as the value of the expression **a=b** is the new value of **a**.

Like any other expression, **++a** may constitute a statement all by itself: just add a semicolon. For example, here are pairs of statements that do the same thing:

```
1 a = a + 1;
2 ++a;

3 for (i = 1; i <= 10; i = i + 1) {
4 for (i = 1; i <= 10; ++i) {

5 for (i = 10; i >= 1; i = i - 1) {
6 for (i = 10; i >= 1; --i) {
```

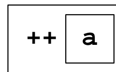
++a may also be part of a larger expression. For example, you can change each example on the left to the one on the right.

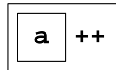
```
1 ++a;
2 printf("%d\n", a);                  printf("%d\n", ++a);

3 ++a;
4 b = a;                              b = ++a;

5 ++a;
6 c[a] = d;                           c[++a] = d;

7 ++a;
8 if (a == c) {                       if (++a == c) {
```

Prefix vs. postfix: K&R pp. 46–47; King p. 54

Most unary operators are written before their operand: **-a**, **&a**, **sizeof a**, etc. The unary operator **++**, however, can be written either before or after its operand:

 All rights reserved ©2007 Mark Meretzky





Both of the above expressions have the same side effect: they both add 1 to **a**. Thus each statement in the following trios does the same thing:

```
1 a = a + 1;
2 ++a;
3 a++;

4 for (i = 1; i <= 10; i = i + 1) {
5 for (i = 1; i <= 10; ++i) {
6 for (i = 1; i <= 10; i++) {

7 for (i = 10; i >= 1; i = i - 1) {
8 for (i = 10; i >= 1; --i) {
9 for (i = 10; i >= 1; i--) {
```

But the expressions **++a** and **a++** have different values. This doesn't matter in the above examples, because **++a** and **a++** did not have their values used as part of a larger expression. The value of **++a** is the new value of **a** (after the increment); the value of **a++** is the old value of **a** (before the increment).

When the computer evaluates the expression **a++**, you can imagine that three things happen in the following order:

(1) The value of **a** is stored in an invisible variable.

(2) 1 is added to the variable **a**.

(3) If the **a++** is a subexpression of a larger expression, the invisible variable is used as the value of the subexpression **a++**.

For example,

```
1     a = 10;
2     b = ++a;                      /* preincrement example */
3     printf("%d %d\n", a, b);
```

```
11 11
```

```
4     a = 10;
5     b = a++;                      /* postincrement example */
6     printf("%d %d\n", a, b);
```

```
11 10
```

```
7     a = 10;
8     printf("%d\n", ++a);          /* preincrement example */
9     printf("%d\n", a);
```

 All rights reserved ©2007 Mark Meretzky

```
11
11
```

```
10     a = 10;
11     printf("%d\n", a++);            /* postincrement example */
12     printf("%d\n", a);
```

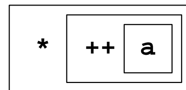
```
10
11
```

You can change each example on the left to the one on the right:

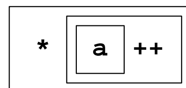
```
 1 printf("%d\n", a);                  printf("%d\n", a++);
 2 a++;

 3 b = a;                              b = a++;
 4 a++;

 5 c[a] = d;                           c[a++] = d;
 6 a++;

 7 if (a == c) {                       if (a++ == c) {
 8     a++;                                blah blah blah;
 9     blah blah blah;                 } else {
10 } else {                                bleh bleh bleh;
11     a++;                            }
12     bleh bleh bleh;
13 }
```

Associativity of unary operators

In the expression `*++a`, the `++` is obviously executed before the `*` because it has elbowed its way closer to the `a`. You don't need the table on K&R p. 53; King p. 595:



But in the expression `*a++`, the two unary operators are equidistant from the `a`. Which one executes first? Line 2 of the table on K&R p. 53; King p. 595 shows that all the unary operators associate from right to left:



If you want the computer to execute the `*` before the `++` in the expression `*a++`, use parentheses: `(*a)++`.

▼ Homework 2.13: draw the boxes

 All rights reserved ©2007 Mark Meretzky

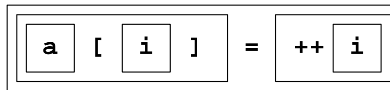
```
*a++                                    ++*a

*(a++)                                  -*++a

(*a)++                                  **a

-!*a++                                  *a++ = *b++
```

Unpredictable values

Never apply the `++` or `--` operators to an expression that appears more than once in a larger expression. For example, the expression `i` appears more than once in the expression `a[i] = ++i`.

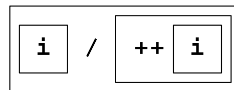
```
1     int a[10];
2     int i = 5;
3
4     a[i] = ++i;   /* Put 6 into a[5] on some platforms, into a[6] on others. */
```

Some machines will evaluate the left operand `a[i]` of `=` before the right operand `++i` of the `=`, storing `6` into `a[5]`. Other machines will evaluate the right operand `++i` of `=` before the left operand `a[i]` of the `=`, storing `6` into `a[6]`.




```
5     int i = 10;
6
7     printf("%d\n", i/++i);
```

Some machines will evaluate the numerator `i` before the denominator `++i`, `printf`'ing `10/11`, or `0`. Other machines will evaluate the denominator `++i` beforethe numerator `i`, `printf`'ing `11/11`, or `1`.



▼ Homework 2.14: lvalues

The left operand of `=` `+=` `-=` etc., and the operand of `++` `--` must be an lvalue. Which of the following are not?

```
a = 10                                  ++a
10 = a                                  ++10
a += 10                                 a + b = 10
10 += a                                 ++(a + b)
```

The data type of an expression: K&R pp. 42–46, 198; King pp. 124–127

Each variable and constant has a data type. See K&R pp. 37, 192–4; King pp. 112, 115–116 for the data type of a constant.

```
1     int i;
2     double d;
3
```

 All rights reserved ©2007 Mark Meretzky

```
 4      /* i and 10 are int's; d and 3.14159 are double's. */
 5      printf("%d %d %f %f\n", i, 10, d, 3.14159);
```

A more complicated expression has a data type, too. You need to know the data type of an expression if you want to **printf** it. The data type of an expression is determined by the data types of its subexpressions:

```
 6      int i, j;
 7      double d, e;
 8
 9      /* The expressions i+j and -i are of type int. */
10      printf("%d %d\n", i + j, -i);
11
12      /* The expressions d+e and -d are of type double. */
13      printf("%f %f\n", d + e, -d);
```

But what is the data type of the expression **i+d** in

```
14      int i;
15      double d;
16
17      printf("%f\n", i + d);
```

All computers have a machine language instruction to add two **int**'s yielding an **int** result, and another instruction to add two **double**'s yielding a **double** result. No computer, however, has an instruction to add an **int** and a **double**.

When you ask the computer to add **i+d**, it creates a copy of the value of **i** converted to data type **double**, and then adds this copy to **d**. The result is of data type **double** (because both addends were of data type **double**), which is why we **printf** the expression **i+d** with **%f**.

In general, when you add two expressions (e.g., **a+b**) the values that are added may not be the values of the two expressions that you wrote. The computer may make copies of one or both of your values, converted as above to larger data types, and it will be the copies that are actually added together.

Ditto for all the operations: subtraction, multiplication, etc. For simplicity, I will write "addition" to mean any operation in the following simplification of the rules in the book.

(1) If the operands are all smaller than data type **int**, then the result will be of data type **int**. For example, a **char** plus a **char** will yield an **int** sum, and a **char** plus a **short** will yield an **int** sum.

(2) Otherwise if all the operands are the same data type, the result will also be that same data type. For example, an **int** plus an **int** will yield an **int** sum, and a **double** plus a **double** will yield a **double** sum.

(3) Otherwise the operands are of different data types, and the result will be of whichever data type is larger. For example, an **int** plus a **double** will yield an **double** sum.

The innermost boxes are evaluated first, so in the expression

```
1      char c;
2      short s;
3      double d;
4
5      printf("%f\n", c * s + d);
```

the computer evaluates the expression **c*s**, yielding an **int** value, before it begins to tackle the addition. Then it makes a **double** copy of that **int** value, and adds this copy to **d**. The value resulting from the addition of these two **double** values is, of course, a **double**.

 All rights reserved ©2007 Mark Meretzky

▼ Homework 2.15: what is the data type of these expressions?

The expression **`100`** is of data type **`int`**; the expression **`100.0`** is of data type **`double`** (K&R pp. 37, 194; Ki pp. 115–116).

```
char c;
short s;
int i, j;
long tallsally;
float f, g;
double d, e;

i / j                                i / 100
i / d                                i / 100.0
c + s                                f + g
c * d                                100 * i / j
i + tallsally                        100.0 * i / j
```

Print a double in scientific notation: K&R pp. 154, 244; King p. 115

```
1     double d = 123.4567;
2
3     printf("%f\n", d);
4     printf("%e\n", d);
```

```
123.4567
1.234567e+02      /* 1.234567 × 10^2 */
```

Casts: K&R pp. 45–6, 198, 205; King pp. 128–129

Line 7 below shows that the two operands of an **`=`** can be two different data types. It deposits a copy of the value of the right operand, converted to a new data type, into the left operand.

The **`(double)`** in line 9 is a unary operator called a *cast*. There is one unary operator for each data type: **`(char)`**, **`(short)`**, **`(int)`**, etc. The value of the expression **`(double)i`** in line 9 is the value of **`i`** converted to type **`double`**. The cast operator has no side effect: it does not change the value of any variable.

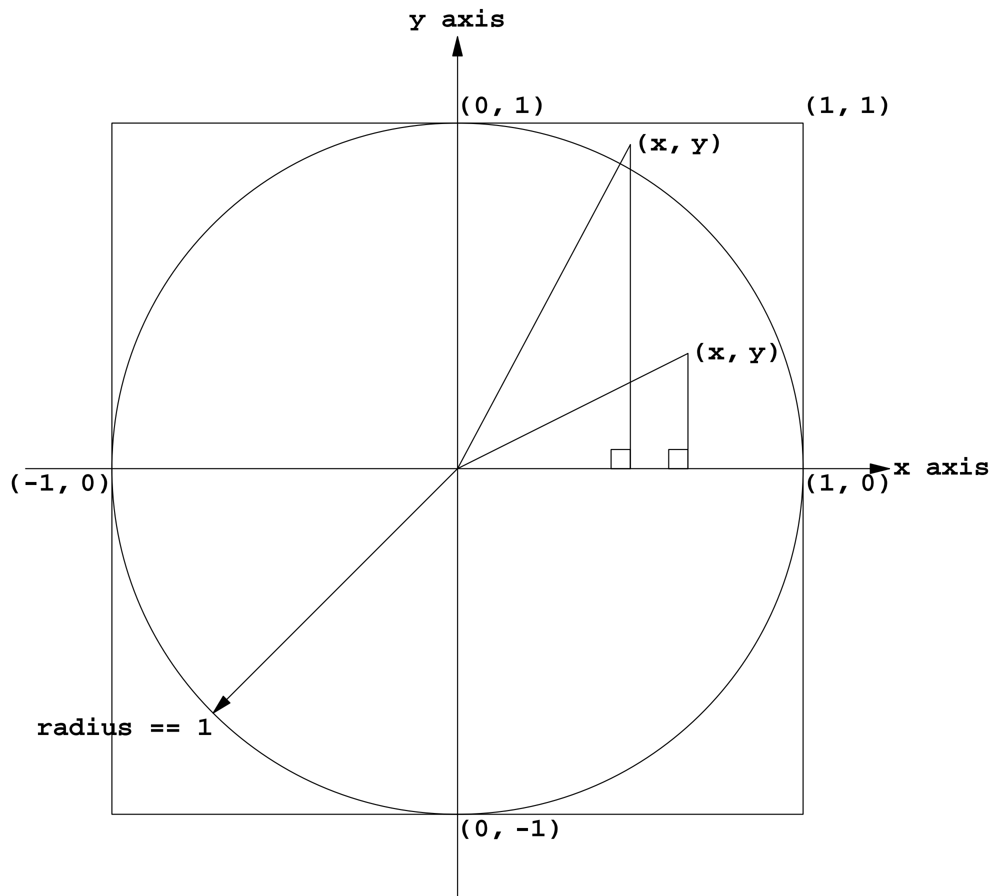
The parentheses around the expression **`i+j`** in line 14 below force the binary operator **`+`** to be evaluated before the unary operator **`(double)`**. See the table on K&R p. 53; King p. 595.

```
 1 int i = 100
 2 int j = 200;
 3 double d;
 4
 5 printf("%d\n", i);                          /* It prints 100 */
 6
 7 d = i;
 8 printf("%e\n", d);                          /* It prints 1.000000e+02 */
 9 printf("%e\n", (double)i);                  /* It prints 1.000000e+02 */
10
11 printf("%d\n", i + j);                      /* It prints 300 */
12 d = i + j;
13 printf("%e\n", d);                          /* It prints 3.000000e+02 */
14 printf("%e\n", (double)(i + j));            /* It prints 3.000000e+02 */
15
```

 All rights reserved ©2007 Mark Meretzky

```
16 printf("%d\n", i / j);                          /* It prints 0 */
17 printf("%f\n", (double)i / j);                  /* It prints 0.500000 */
18 printf("%f\n", i / (double)j);                  /* It prints 0.500000 */
19
20 printf("%f\n", (double)i / (double)j);          /* It prints 0.500000 (overkill!) */
21 printf("%f\n", (double)(i / j));                /* It prints 0.000000 (too late!) */
```

▼ Homework 2.16: Compute the value of π



The area of a circle is

$$A = \pi r^2$$

The radius of this circle is 1, so its area is π. The area of the big square is 4. As you can see, the circle occupies approximately 80% of the square, so the area of the circle (and hence the value of π) is approximately 80% of 4, or 3.2. Find this percentage more exactly by the Monte Carlo method to be described in class.

Call **`srand`** so you get different random numbers each time you run the program. Print one line of column headings.

Then write a **`for`** loop that iterates 1000 times. Use an **`int`** variable named **`i`** to count from 1 to 1000. During each iteration, call the **`rand`** function and divide its return value by the largest possible random number **`RAND_MAX`**, yielding a random number between 0 and 1.

Remember to **`#include <stdlib.h>`** as in Handout 1, p. 23. This gives you the right to use the macro **`RAND_MAX`** (K&R p. 252; King p. 570); use it as if it were a variable.

 All rights reserved ©2007 Mark Meretzky

Store the random number between 0 and 1 in a **double** variable named **x**. Store another random number between 0 and 1 and store it in a **double** variable named **y**. **printf** the values of **x** and **y** to make sure that they are random numbers between 0 and 1; that they change during each iteration; and that they are almost never equal to each other. Do not write the rest of the program until this part works. When it does, remove (or comment out) the **printf** before handing it in.

Then compute $\sqrt{\mathbf{x}^2 + \mathbf{y}^2}$. Since C has no exponentiation, you'll have to write **x*x** and **y*y**. **#include <math.h>** to declare the **sqrt** function for you. Count how many points lie within the circle with an **int** variable named **count**. You get no credit if **int** and **count** are not **int**'s. You get no credit if you forget to initialize **count** to **0** before the first iteration. Can you eliminate the square root by squaring both sides of the inequality? If so, you no longer need to **#include <math.h>**.

During the 100th, 200th, 300th, etc., iteration, **printf** the total number of points, the number of points within the circle, the approximate value of π, and the error. In other words, output only ten lines even though the loop iterates 1000 times. Do not execute this **printf** during every loop. use The approximate value of π is

$$4 \times \frac{\textit{number within}}{\textit{total number}}$$

If you divide two expressions of data type **int**, the result will be another expression of data type **int**: the fractional part of the quotient will be chopped off. Avoid this by making either the numerator or the denominator an expression of data type **double**. Instead of using an explicit **(double)** cast, it would be simpler to use the fact that the constant **4** is of data type **int**, and the constant **4.0** is of data type **double**.

The error is

$$4 \times \frac{\textit{number within}}{\textit{total number}} - 3.14159265358979323846$$

But don't write the 3.14159265358979323846 in the middle of the program—make a **const** variable for it:

```
const double pi = 3.14159265358979323846;     /* true value of pi */
```

Don't write the numbers **1000** or **100** anywhere except in **const** variables. You'll have to invent names for them.

Print the error with **%9.6f** instead of **%f** to make the decimal points line up; see K&R pp. 12–13. To avoid the **undefined sqrt** error message from the linker, you may have to compile with the "math library" option **-lm**:

```
1$ gcc -o pi pi.c -lm                                  minus lowercase LM
2$ pi
total        within            pi         error
  100            83      3.320000      0.178407
  200           163      3.260000      0.118407
  300           234      3.120000     -0.021593
  400           303      3.030000     -0.111593
  500           381      3.048000     -0.093593
  600           462      3.080000     -0.061593
  700           545      3.114286     -0.027307
  800           624      3.120000     -0.021593
  900           703      3.124444     -0.017148
 1000           781      3.124000     -0.017593
```

Make a **double** variable named **approximate** to hold your approximate value of π. Do not compute the approximate value of π during every iteration—compute it only when you're going to **printf** it (i.e., during only one out of every 100 iterations of the **for** loop). You get no credit if you compute it more often than you print it. Your program must have one **for** loop, two **printf**'s, two **if**'s, and no **else**'s;

 All rights reserved ©2007 Mark Meretzky

you get no credit otherwise. Your program must have no non-**const** variables other than **count**, **i**, **x**, **y**, and **approximate**; you get no credit otherwise.

✎ Extra credit. Instead of iterating 1000 times and printing results at intervals of 100 (the ten lines of output shown above), iterate 100,000 times and print the results after 1 iteration, 10 iterations, 100 iterations, 1,000 iterations, 10,000 iterations, and 100,000 iterations (six lines of output). Change the two **int** variables to **long**. Use an additional **long** variable named **power** whose values will run through the series of numbers **1**, **10**, **100**, **1000**, **10000**, **100000**, and a **const long** variable named **factor** whose value will be 10. No credit if the series **1**, **10**, **100**, **1000**, **10000**, **100000** is written in your program.

Don't tell them what they can read for themselves

```
 1 /*
 2 Program Function: Compute the value of pi.
 3 Program Description: This program will loop 1000 times. During
 4 each loop the function rand() will be called twice. The two random
 5 number will be put into the variables X and Y. X and Y will be divided
 6 by the largest possible random number, yielding two random numbers between
 7 0 and 1. These two numbers will then be put back into X and Y, replacing
 8 the old values.  A random number will be seeded using srand() with
 9 the time. Please note, this program was run on with Borland/C,
10 using the it's variation of the time function */
11
12 #define N 1000
```

```
 1 /* METHODOLOGY */
 2 /* add 1 to the day value, if value > last day of that month,*/
 3 /* then increase month by 1 and change day to 1, if that */
 4 /* month > 12, then increase year value by one, and change */
 5 /* day to 1, and month to 1 and you're done...*/
```

Tell them the non-obvious

```
 1 /* Compute the value of pi by using the Monte Carlo method to measure the area
 2 of a unit circle (i.e., a circle of radius 1 whose area is therefore pi).
 3
 4 Pick N random pairs of numbers (x, y), where each number is between 0 and 1.
 5 They represent as N random points in a one-by-one square whose lower left corner
 6 is at the origin (i.e., the place where the x and y axis cross).
 7
 8 Count how many of these points lie within a distance of 1 from the origin.  Use
 9 the Pythagorean theorem (c squared == a squared + b squared) to compute this
10 distance.  The percentage of points within this distance is the same as the
11 percentage of the one-by-one square covered by the upper right quarter of the
12 circle.  Now that we have estimated the area of one quarter of the circle,
13 multiply by 4 to compute the area of the entire circle.
14
15 Output a table showing intermediate results after every N/10 random points.
16 Runs under Borland C; other versions of C may use a different time function. */
17
18 #define N 1000          /* the number of random points */
```

Do not copy the above comment into your own program.

 All rights reserved ©2007 Mark Meretzky

How to write an equation:

Suppose you have to write

$$a = \frac{b \times c}{d}$$

The following statements both perform the assignment. What is the best way to make it clear which variables are in the numerator and which are in the denominator?

```
a = b * c / d;        /* straightforward */
a = b / d * c;        /* correct, but misleading */
```

$$a = \frac{b \times c}{d \times e}$$

```
a = b / d * c / e;       /* ignores the above advice */
a = b * c / d / e;       /* a little clearer */
a = b * c / (d * e);     /* much clearer and faster */
```

□

 All rights reserved ©2007 Mark Meretzky