Fall 2006 Handout 5

The comma operator: K&R pp. 62−63, 209; King pp. 94−94

You're allowed to write only at most one expression in each of the three sections of a **`for (;;)`** (line 2):

```
1     d = dest;
2     for (s = source; s < source + N; ++s) {
3         do something;
4         ++d;
5     }
```

If you'd like to write two expressions in a place that allows only one, use the binary operator **`,`** to join two expressions into one big expression. The left subexpression is evaluated before the right subexpression.

```
6     for (s = source, d = dest; s < source + N; ++s, ++d) {
7         do something;
8     }
```

Use the comma operator only in the first and third sections of a **`for (;;)`**

Copying char's: K&R pp. 105−6

Here are five ways to copy the first **`N`** bytes from an array **`source`** to an array **`dest`**. We always copy **`N`** bytes, even if we encounter a **`'\0'`** byte.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>      /* for memcpy */
 4
 5 #define SIZE     256     /* number of elements in each array */
 6 #define  N       10      /* number of bytes to copy */
 7
 8 main()
 9 {
10     char source[SIZE];
11     char dest[SIZE];
12
13     int i;               /* index */
14
15     char *s;             /* pointer into source[] */
16     char *d;             /* pointer into dest[] */
17
18     /* Method 1 */
19     for (i = 0; i < N; ++i) {
20         dest[i] = source[i]; /* Copy one byte from source to dest. */
21     }
22
23     /* Method 2 */
24     for (s = source, d = dest; s < source + N; s++, d++) {
```

```
25          *d = *s;         /* Copy one byte from source to dest. */
26      }
27
28      /* Method 3 */
29      s = source;
30      d = dest;
31      while (s < source + N) {
32          *d = *s;
33          s++;
34          d++;
35      }
36
37      /* Method 4 */
38      s = source;
39      d = dest;
40      while (s < source + N) {
41          *d++ = *s++;
42      }
43
44      /* Method 5: K&R p. 250; King pp. 151-152. */
45      memcpy(dest, source, N);
46
47      exit(EXIT_SUCCESS);
48  }
```

Numerical values of relational expressions:
K&R pp. 41−42, 206; King pp. 64−66

Relational expressions such as **`a == b`** are actually numerical expressions. Their value is always of data type **`int`**: **`1`** for true, **`0`** for false.

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  main()
 5  {
 6      int a, b;
 7
 8      printf("Please type two numbers, pressing RETURN after each.\n");
 9      scanf("%d%d", &a, &b);
10
11      printf("%d\n", a == b);      /* Prints 1 or 0. */
12      exit(EXIT_SUCCESS);
13  }
```

The relational expression in control structures such as

```
    if (a == b) {
    while (a == b) {
    for (i = 0; i <= 10; ++i) {
```

is actually a numerical expression. It counts as ‘‘true’’ if its value is *any* non-zero number, and ‘‘false’’ if its vale is zero (p. 56). For example, if **`a`** equals **`b`**, then the value of the expression **`a==b`** is 1 and the following **`if`** is therefore true:

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

Use the numerical value of a relational expression.

The parentheses on lines 26 and 36 are unnecessary: **==** has higher precedence than **+=** in the table on K&R p. 53; King p. 595.

```
 1 /* Print the number of questions the student answered correctly. */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 main()
 6 {
 7     int correct[] = {242, 238, 231, 225, 215, 207};       /* correct answers */
 8     int student[] = {242, 238, 230, 225, 215, 208};       /* student's answers */
 9
10 #define N 6
11
12     int i;
13     int *p, *q;
14     int count = 0;
15
16     /* Five ways to write the same loop. */
17
18     for (i = 0; i < N; ++i) {
19         if (correct[i] == student[i]) {
20             ++count;
21         }
22     }
23
24     for (i = 0; i < N; ++i) {
25         /* count = count + (correct[i] == student[i]); */
26         count += (correct[i] == student[i]);
27     }
28
29     for (p = correct, q = student; p < correct + N; ++p, ++q) {
30         if (*p == *q) {
31             ++count;
32         }
33     }
34
35     for (p = correct, q = student; p < correct + N; ++p, ++q) {
36         count += (*p == *q);
37     }
38
39     printf("There are %d correct answers.\n", count);
40     exit(EXIT_SUCCESS);
41 }
```

This program has two weaknesses. (1) You need two pointers, **p** and **q**, even though you needed only one index **i**. (2) You get a bug if you accidentally made the arrays two different lengths. Both problems will disappear when we rewrite the two arrays as a single ‘‘array of structures’’.

Numerical values of logical expressions:
K&R pp. 41−42, 207−208; King pp. 64−66

The left and right operands of **&&** are usually thought of as logical expressions:


```
a == b && c == d
```

But we now know that **a == b** and **c == d** are actually arithmetic expressions. The value of the expression

```
a == b && c == d
```

is always of data type **int**. The value is 1 if the values of both operands are *any* non-zero numbers; the value is 0 otherwise. Similarly, the value of **||** is 1 if the value of either operand* is *any* non-zero number; the value is 0 otherwise. Finally, the value of the unary operator **!** is 0 if the value of its operand is *any* non-zero number; the value is 1 otherwise.

For example, if **a** equals **b** and **c** equals **d**, then the value of the expression **a==b** is 1, the value of the expression **c==d** is 1, the value of the expression **a==b && c==d** is 1, and the following **if** is therefore true:

```
if (a == b && c == d) {
```

Short circuiting: K&R pp. 41−42, 52; King p. 66

In the expression

```
a * b + c * d
```

both multiplications are always performed, but it is impossible to predict which one is evaluated first. (No one knows or cares.) But in the expression

```
a == b && c == d
```

both comparisons are not always performed, and it is possible to predict which one is evaluated first when both are performed.

The left operand of **&&** is always evaluated first. The right operand is evaluated only if the left one was non-zero. If the left operand is zero, there is no point in proceeding—we already know the answer.

Similarly, the left operand of **||** is always evaluated first. The right operand is evaluated only if the left one was zero. If the left operand is non-zero, there is no point in proceeding—we already know the answer.

Use && to avoid nested if's

```
 1     /* other languages: */
 2     if (b != 0) {
 3         if (a / b == c) {
 4             printf("The quotient is c.\n");
 5         }
 6     }
 7
 8     /* C: */
 9     if (b != 0 && a / b == c) {
10         printf("The quotient is c.\n");
11     }
12
13     /* Other languages: */
14     if (i >= 0 && i < N) {
15         if (a[i] == b) {
16             printf("a[i] equals b.\n");
17         }
18     }
```

* or both operands

```
19
20      /* C: */
21      if (i >= 0 && i < N && a[i] == b) {
22          printf("a[i] equals b.\n");
23      }
24
25      /* Other languages: */
26      if (p != NULL) {
27          if (*p != '\0') {
28              printf("The string has at least one character in it.\n");
29          }
30      }
31
32      /* C: */
33      if (p != NULL && *p != '\0') {
34          printf("The string has at least one character in it.\n");
35      }
```

Use || to avoid repeating the same body

```
 1      /* inelegant */
 2      if (a == b) {
 3          printf("At least one pair is equal.\n");
 4      } else if (c == d) {
 5          printf("At least one pair is equal.\n");
 6      }
 7
 8      /* elegant */
 9      if (a == b || c == d) {
10          printf("At least one pair is equal.\n");
11      }
```

Compare Handout 1, pp. 30−31:

—On the Web at
http://i5.nyu.edu/~mm64/x52.9232/src/leap2.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 % 4 == 0 && (year % 100 != 0 || year % 400 == 0)) {
11         printf("%d is a leap year.\n", year);
12     } else {
13         printf("%d is not a leap year.\n", year);
14     }
15 }
```

The replacement text of a macro does not have to be a number.

```
 1     int c1;      /* catalog numbers */
 2     int c2;
 3     int c3;
 4     int i;       /* loop counter to draw row of dashes */
 5
 6     printf("%d", c1);

 7 #define CATNO    int
 8 #define FORMAT   "%d"
 9
10     CATNO c1;
11     CATNO c2;
12     CATNO c3;
13     int i;       /* loop counter to draw row of dashes */
14
15     printf(FORMAT, c1);
```

If you migrate to a platform whose **int**'s are not big enough to hold a catalog number, you can change lines 7−8 to

```
16 #define CATNO    long
17 #define FORMAT   "%ld"    /* percent lowercase LD */
```

Three rules for writing #define's: K&R pp. 89−90; King pp. 277−288

(1) If the **#define** expands into two or more tokens, put parentheses around them.

```
 1 #define  N   280              /* number of days in the pregnancy */
 2 #define  N   (10 * 28)        /* ten lunar months */
 3 #define  EOF (-1)             /* end of file (p. 16): see /usr/include/stdio.h */
 4
 5 #define  TAX 1020
 6 #define  TAX (610+395+15)     /* federal, state, city */
```

What goes wrong in line 12 if line 7 omits the parentheses of line 6?

```
 7 #define  TAX (610+395+15)     /* federal, state, city */
 8     int oldtax;               /* last year's */
 9     int newtax;               /* this year's */
10
11     oldtax = TAX;
12     newtax = 2 * TAX;         /* My taxes doubled this year. */
```

What two morals does the following example teach?

```
13 #define  b   -10
14
15     printf("%d\n" a-b);
```

(2) If the **#define** expands into two or more statements, put curly brackets around them.

```
 1 #define  DEBUGX       printf("%d\n", x)
 2 #define  DEBUGXY      {printf("%d\n", x); printf("%d\n", y);}
```

What goes wrong in line 17 if line 4 omits the parentheses of line 2?

```
 3 #define  DEBUGX       printf("%d\n", x)
```

```
 4 #define DEBUGXY     {printf("%d\n", x); printf("%d\n", y);}
 5
 6     int x, y;
 7
 8     if (x < 0) {
 9         DEBUGX;
10     }
11
12     if (x != y) {
13         DEBUGXY;
14     }
15
16     if (x < 0) DEBUGX;
17     if (x != y) DEBUGXY;
```

(3) Parenthesize every argument in a **#define** line.

```
1 #define SQUARE(x)   ((x) * (x))
```

What goes wrong in line 8 if line 2 omits the inner parentheses of line 1? **x**?

```
1 #define SQUARE(x)   (x * x)
2
3     int i = 2;
4     int j = 3;
5
6     printf("%d\n", SQUARE(i));      /* It prints 4. */
7     printf("%d\n", SQUARE(i + j));  /* It prints 11; should be 25. */
```

A useful macro that requires parentheses

You can omit the dimension from the square brackets in an array initialization (p. 86). This saves you the trouble of counting the list of values.

```
char s[] =              /* Handout 4, pp. 3-4, lines 1-3 */
int correct[] = {       /* Handout 5, p. 3, line 7 */
```

Unfortunately, the omitted number often has to appear elsewhere in the program. For example, even if we remove the number **6** from the square brackets in the array in Handout 5, p. 3, line 7, we would still have to write **6** in the macro definition in line 10.

The expression **sizeof correct** is the number of bytes in the entire **correct** array. The expression **sizeof correct[0]** is the number of bytes in the first element of the array. Their quotient is the number of elements in the array.

This works for all data types on all machines. It will make **N** change its value automatically if you add new array elements: you no longer have to count them yourself and type their number. See K&R pp. 132−136; King pp. 338−339 for a more complicated example with an array of structures.

```
6 int correct[] = {242, 238, 231, 225, 215, 207};
7
8 /* number of elements in the correct array */
9 #define N   (sizeof correct / sizeof correct[0])
```

An example of a macro with an argument

```
1     double ffreeze = 32;                    /* freezing point in Fahrenheit*/
2     double fboil = 212;                     /* boiling point in Fahrenheit */
3
```

```
 4     double cfreeze = (ffreeze - 32.0) * 5.0 / 9.0; /* freezing point in celsius */
 5     double cboil   = (fboil   - 32.0) * 5.0 / 9.0; /* freezing point in celsius */

 6 /* Return the celsius temperature corresponding to the given fahrenheit. */
 7 #define CELSIUS(f)   (((f) - 32.0) * 5.0 / 9.0)
 8
 9     double ffreeze = 32;                        /* freezing point in Fahrenheit */
10     double fboil = 212;                         /* boiling point in Fahrenheit */
11
12     double cfreeze = CELSIUS(ffreeze);
13     double cboil   = CELSIUS(fboil);
```

An example of a macro with an argument

```
 1     /* an array whose subscripts range from -10 to 10 inclusive */
 2     int a[21];
 3 #define A(i) (a[(i) + 10])
 4
 5     for (i = -10; i <= 10; ++i) {
 6         printf("%d\n", A(i));
 7     }
```

Better yet,

```
 8     /* an array whose subscripts range from -N to N inclusive */
 9 #define N 10
10     int a[2 * N + 1];
11 #define A(i) (a[(i) + N])
12
13     for (i = -N; i <= N; ++i) {
14         printf("%d\n", A(i));
15     }
```

32-bit signed and unsigned integer values

(signed) **int**	*bit pattern*	**unsigned**	*bit pattern*
		4294967295	**11111111111111111111111111111111**
		4294967294	**11111111111111111111111111111110**
		4294967293	**11111111111111111111111111111101**
		4294967292	**11111111111111111111111111111100**
		4294967291	**11111111111111111111111111111011**
		4294967290	**11111111111111111111111111111010**
		2147483653	**10000000000000000000000000000101**
		2147483652	**10000000000000000000000000000100**
		2147483651	**10000000000000000000000000000011**
		2147483650	**10000000000000000000000000000010**
		2147483649	**10000000000000000000000000000001**
		2147483648	**10000000000000000000000000000000**
2147483647	**01111111111111111111111111111111**	2147483647	**01111111111111111111111111111111**
2147483646	**01111111111111111111111111111110**	2147483646	**01111111111111111111111111111110**
2147483645	**01111111111111111111111111111101**	2147483645	**01111111111111111111111111111101**
2147483644	**01111111111111111111111111111100**	2147483644	**01111111111111111111111111111100**
2147483643	**01111111111111111111111111111011**	2147483643	**01111111111111111111111111111011**
2147483642	**01111111111111111111111111111010**	2147483642	**01111111111111111111111111111010**
5	**00000000000000000000000000000101**	5	**00000000000000000000000000000101**
4	**00000000000000000000000000000100**	4	**00000000000000000000000000000100**
3	**00000000000000000000000000000011**	3	**00000000000000000000000000000011**
2	**00000000000000000000000000000010**	2	**00000000000000000000000000000010**
1	**00000000000000000000000000000001**	1	**00000000000000000000000000000001**
0	**00000000000000000000000000000000**	0	**00000000000000000000000000000000**
−1	**11111111111111111111111111111111**		
−2	**11111111111111111111111111111110**		
−3	**11111111111111111111111111111101**		
−4	**11111111111111111111111111111100**		
−5	**11111111111111111111111111111011**		
−6	**11111111111111111111111111111010**		
−2147483643	**10000000000000000000000000000101**		
−2147483644	**10000000000000000000000000000100**		
−2147483645	**10000000000000000000000000000011**		
−2147483646	**10000000000000000000000000000010**		
−2147483647	**10000000000000000000000000000001**		
−2147483648	**10000000000000000000000000000000**		

The bits of an integer are numbered from right to left, starting at 0. In a 32-bit integer, the leftmost bit is therefore bit 31. Bit 0 is called the *lowest* bit or the *low order* bit. Bit 31 is called the *top* bit or the *high order* bit.

Bit 0 is the *one's place,* bit 1 is the *two's place,* bit 2 is the *four's place,* etc. In signed values, the top bit is called the *sign bit.* It is 1 for negative numbers, 0 otherwise.

Why is the integer −1 represented as 11111111111111111111111111111111 (thirty-two 1's)?

Think of a car odometer running backward:


```
0003
0002
0001
0000        zero
9999        negative one
9998        negative two
```

1111 is the binary equivalent of 9999.

If you want zero to be **00000000000000000000000000000000**, positive one to be **00000000000000000000000000000001**, and positive one plus negative one to be 0, then the definition of negative one is forced on you:

```
  00000000000000000000000000000001   positive one
+ 11111111111111111111111111111111   negative one
  00000000000000000000000000000000   zero

  00000000000000000000000000000010   positive two
+ 11111111111111111111111111111110   negative two
  00000000000000000000000000000000   zero

  00000000000000000000000000000011   positive three
+ 11111111111111111111111111111101   negative three
  00000000000000000000000000000000   zero

  00000000000000000000000000000100   positive four
+ 11111111111111111111111111111100   negative four
  00000000000000000000000000000000   zero
```

```
1     int i = ~0;              /* all ones: KR p. 49 */
2     printf("%d\n", i);       /* prints -1 */
3     printf("%u\n", i);       /* prints 4294967295 */

4     char c = ~0;             /* all ones */
5     int i = c;
6     printf("%d\n", i);       /* -1 if char is signed, 255 if char is unsigned */
```

Sign extension: K&R p. 198

Thanks to sign extension, **i1** equals **s1**, **i2** equals **s2**, and **i3** equals **us**. If there were no sign extension, the value of **i2** in line 5 would be 65,535 instead of −1.

```
1     short s1 = 1;                  /* binary                  0000000000000001 */
2     int i1 = s1;                   /* binary 00000000000000000000000000000001 */
3
4     short s2 = -1;                 /* binary                  1111111111111111 */
5     int i2 = s2;                   /* binary 11111111111111111111111111111111 */
6
7     unsigned short us = 65535;     /* binary                  1111111111111111 */
8     int i3 = us;                   /* binary 00000000000000001111111111111111 */
```

Left shift and right shift: K&R pp. 49, 206; King p. 452

Here are four ways to double the value of **i**:

```
1     int i = 10;         /* 00000000000000000000000000001010 */
```



```
 2
 3      i = i * 2;          /* 00000000000000000000000000010100 */
 4      i *= 2;
 5
 6      i = i << 1;         /* << executes faster than * */
 7      i <<= 1;
```

Here are four ways to halve the value of **i**:

```
 8      int i = 10;         /* binary 00000000000000000000000000001010 */
 9
10      i = i / 2;          /* binary 00000000000000000000000000000101 */
11      i /= 2;
12
13      i = i >> 1;         /* >> executes faster than / */
14      i >>= 1;
```

(1) When left shifting any data type, zeroes enter from the right as shown above.

(2) When right shifting **int**, **long**, **short**, or **signed char**, the sign bit propagates (i.e., extra copies of the sign bit enter from the left):

```
15      unsigned u = -4;    /* binary 11111111111111111111111111111100 */
16      u >>= 1;            /* binary 11111111111111111111111111111110 */
17      /* Now u == -2. */
```

(3) When right shifting **unsigned**, **unsigned long**, **unsigned short** or **unsigned char**, zeroes enter from the left.

Bitwise "and", "or", "exclusive or", and "not": K&R pp. 48−49, 204, 207; King pp. 453−457

Addition and subtraction are complicated because there is carrying or borrowing. For example,

```
    0  0  1  1
+   0  1  0  1
-------------
    1  0  0  0
```

Bitwise ''and'' and ''or'' etc. are simpler than addition and subtraction because there is no carrying or borrowing. Each place is independent of the adjacent places: It's just like 16- or 32-track stereo. For example, bitwise ''and'' yields a 1 if the two bits above it are 1, and yields 0 otherwise:

```
    0  0  1  1
&   0  1  0  1
-------------
    0  0  0  1
```

Bitwise ''or'' yields a 1 if either (or both) of the two bits above it are 1, and yields 0 otherwise:

```
    0  0  1  1
|   0  1  0  1
-------------
    0  1  1  1
```

Bitwise ''exclusive or'' yields a 0 if the two bits above it are the same, and yields 1 otherwise:

```
    0  0  1  1
^   0  1  0  1
-------------
    0  1  1  0
```

Bitwise ''one's complement'' yields a 1 if the bit above it is 0, and yields 1 otherwise:

```
~   0   1
    1   0
```

```
1     /* Assume int's are 4 bytes. */
2     int i = 1;          /* binary 00000000000000000000000000000001 */
3     int j = 2;          /* binary 00000000000000000000000000000010 */
4     int k = i & j;      /* binary 00000000000000000000000000000000 */
5
6     k = i | j;          /* binary 00000000000000000000000000000011 */
7     k = i ^ j;          /* binary 00000000000000000000000000000011 */
8     k = ~i;             /* binary 11111111111111111111111111111110 */
```

Turn one bit on with "or"

```
1     char c = 'A';       /* Put the byte 01000001 into c (ASCII code of 'A'). */
2
3     c = c | 32;         /* Put the byte 01100001 into c (ASCII code of 'a'). */
4     c |= 32;            /* A better way to write line 3. */
5     c |= (1 << 5);      /* An even better way to write line 3. */
```

```
    0100 0001          'A'
|   0010 0000          32
    0110 0001          'a'
```

You can write **0x20**, **040**, or **(1<<5)** instead of **32**, but **(1<<5)** is the best because it shows you which bit is on. The parentheses are unnecessary in line 5 in the above example; see K&R p. 53; King p. 595.

Turn one bit off with "and"

```
1     char c = 'a';       /* Put the byte 01100001 into c (ASCII code of 'a'). */
2
3     c = c & 223;        /* Put the byte 01000001 into c (ASCII code of 'A'). */
4     c &= 223;           /* A better way to write line 3. */
5     c &= ~(1 << 5);     /* An even better way to write line 3. */
```

```
    0110 0001          'a'
&   1101 1111          223
    0100 0001          'A'
```

You can write **0xDF**, **0xdf**, **0337**, or **~(1<<5)** instead of **223**, but **~(1<<5)** is the best because it shows which bit is off. The parentheses are necessary in line 5 of the above example.

Thus any bit in any **char**, **int**, **short**, **long**, **unsigned short**, **unsigned**, or **unsigned long** can be turned on or off individually. You can also turn groups of bits on and off together: the example in paragraph 3 of K&R p. 49 turns off the six low bits of **x**. See the remarks there about independence of word length.

Hide the ugliness with macros with arguments.

```
1     char c, d, e;
2     int i, j, k
3
4     c |= (1 << 5);              /* Turn on bit 5 of c. */
5     d |= (1 << 6);              /* Turn on bit 6 of d. */
6     e |= (1 << 7);              /* Turn on bit 7 of e. */
```

```
 7
 8      i &= ~(1 << 8);                 /* Turn off bit 8 of i. */
 9      j &= ~(1 << 9);                 /* Turn off bit 9 of j. */
10      k &= ~(1 << 10);                /* Turn off bit 10 of k. */
11
12      if (((c >> 2) & 1) == 1) {      /* True if bit 2 of c is 1. */
13      if (((d >> 3) & 1) == 0) {      /* True if bit 3 of d is 0. */
14      if (((e >> 4) & 1) != 0) {      /* True if bit 4 of e is not 0. */
```

Don't write the **|=**'s and **&= ~**'s directly: hide them in **#define**'s. I wish we could make a **#define** with two holes in it, for the insertion of a different variable and a different bit position each time it is used:

```
1 /* Turn on a bit in a variable. */
2 #define TURNON      (_____ |= (1 << _____))
```

If the first argument of the following **#define**'s is a **char**, the second argument must be a number in the range 0 to 7 inclusive. If the first argument of the following **#define**'s is an **int**, the second argument must be a number in the range 0 to **8*sizeof(int)-1** inclusive. If the first argument of the following **#define**'s is a **long**, the second argument must be a number in the range 0 to **8*sizeof(long)-1** inclusive.

```
 1 /* Turn on a bit in a variable. */
 2 #define TURNON(variable, bit)        ((variable) |= (1 << (bit)))
 3
 4 /* Turn off a bit in a variable.  Type a tilde: K&R pp. 48-49;
 5 King p. 455  */
 6 #define TURNOFF(variable, bit)       ((variable) &= ~(1 << (bit)))
 7
 8 /* Value of this expression is 1 if the bit is on, 0 otherwise. */
 9 #define TEST(variable, bit)          (((variable) >> (bit)) & 1)
10
11      char c, d, e;
12      int i, j, k;
13
14      TURNON(c, 5);
15      TURNON(d, 6);
16      TURNON(e, 7);
17
18      TURNOFF(i, 8);
19      TURNOFF(j, 9);
20      TURNOFF(k, 10);
21
22      if (TEST(c, 2) == 1) {
23      if (TEST(d, 3) == 0) {
24      if (TEST(e, 4) != 0) {
```

▼ Homework 5.1: redo the binary part of Homework 1.8

Redo the extra credit part of Homework 1.8 (Handout 1, pp. 19−21) using **>>** and **&** instead of **/** and **%**. Or simply use the macro **TEST**. Which is easier to write, and which executes faster? Start from the answer in Handout 3, pp. 16−17.

▲

▼ Homework 5.2: Invent a TOGGLE #define

Invent a **#define** called **TOGGLE** with the same two arguments as **TURNON** and **TURNOFF**. It will complement the specified bit of the specified variable, leaving all the other bits in the variable unchanged. Use ''bitwise exclusive or'', K&R p. 48; King p. 453.

TOGGLE would be used as shown below. Hand in only the **#define** line and its one-sentence comment; do not hand in an example of how **TOGGLE** would be used. You get no credit if **TOGGLE** does not have two arguments.

```
1     int i = 1;
2
3     TOGGLE(i, 3);        /* Now i == 9. */
4     TOGGLE(i, 3);        /* Now i == 1 again. */
```

▲

Prevent sign extension

```
1 #include <stddef.h>
2
3     wchar_t beta = 0x03B2;          /* Unicode lowercase Greek beta */
4
5     printf("%c%c",
6         beta >> 8 & 0xFF,           /* high-order byte */
7         beta & 0xFF                 /* low-order byte */
8     );
```

β

▼ Homework 5.3: add even parity

Write a C program called **parity** that reads input one character at a time. Read K&R pp. 15−17; King pp. 121−122 and use the classic **while-getchar** loop on K&R p. 17 You get no credit unless you make **c** an **int**.

As it reads each character, **parity** should count how many of its lowest seven bits are **1**'s. Do this with a **for** loop that iterates exactly 7 times for each input character.

You get no credit if your **for** loop **TEST**'s or counts the top bit. Do not assume that the top bit of each input character is 0; you get no credit if you rely on this assumption.

Then turn on or turn off the top bit so that the total number of **1**'s will be even. For example, change **01000011** to **11000011**, but leave **01000001** unchanged. Finally, output each character with **putchar**.

Use **TURNON**, **TURNOFF**, and **TEST**. Since this homework and the next both use **TURNON**, **TURNOFF**, and **TEST**, write these **#define**'s in a separate header file called **bit.h** in the same directory as your **.c** files and

```
#include "bit.h"
```

immediately after the **#include**'s that have **<**angle brackets**>**. You get no credit for Homeworks 5.4 or 5.3 unless you hand in **bit.h**. You get no credit for Homeworks 5.4 or 5.3 if any macro in **bit.h** has an odd number of parentheses in its replacement text.

Do not use an array or pointers. Each character should be input, processed, and output before the next character is input. Your program must have exactly one **while**, one **for**, two **if**'s (one of which has an **else**), one **getchar**, one **putchar**, no **printf**'s, no **scanf**'s, and three variables; no credit otherwise. Name them **c** (the ASCII code of each character), **b** (the induction variable of the **for** loop), and **count** (to count how many **1**'s there are in **.c**) You get no credit if you write **count = 0** in more than one place.

Create a short text file called **file1** containing lines such as

```
It is a truth universally acknowledged, that a single man
in possession of a good fortune must be in want of a wife.
```

Feed **file1** to **parity** and collect the output in an output file called **file2**:

```
1$ parity < file1 > file2
2$ od -bvw1 file2 | more                    See an octal dump of file2.
```

Do not attempt to display **file2** directly on the screen or printer: many of the characters it contains will be non-printing. To verify that **parity** worked correctly, feed its output to the next homework. You get no credit for Homeworks 5.3 or 5.4 unless you hand in the output of Homework 5.4.

▲

▼ Homework 5.4: display bytes in binary

Write a C program called **disparity** that will read its input one character at a time like **parity**, and output each character on a line by itself as eight **1**'s and **0**'s. Unlike **parity**, **disparity** must not count the **1**'s and **0**'s.

Run the programs **parity** and **disparity** as follows. Print **file1** and **file3**, but *not* **file2**.

```
1$ parity < file1 > file2
2$ disparity < file2 > file3

3$ lpr -Pth_hp4si_1 parity.c disparity.c file1 file3
4$ rm file1 file2 file3
```

The output in **file3** should begin

```
1 1001001   'I'
0 1110100   't'
1 0100000   ' '
0 1101001   'i'
1 1110011   's'
1 0100000   ' '
```

and should end

```
0 1110111   'w'
0 1101001   'i'
0 1100110   'f'
0 1100101   'e'
0 0101110   '.'
0 0001010   ''          newline—nothing between the single quotes
```

Since **parity.c** and **disparity.c** both use **TURNON**, **TURNOFF**, and **TEST**, write these **#define**'s in a separate header file called **bit.h** in the same directory as your **.c** files and

```
#include "bit.h"
```

disparity must print one blank between the parity bit and the other seven bits. After printing the eight bits, turn off the parity bit. If the character is now in the range 32 to 126 inclusive, **printf** it using **%c** within single quotes as shown above. Otherwise print nothing at all within the single quotes, not even a space. You get no credit for this homework unless every line of output contains a pair of single quotes. Use the last of the following **if**'s:

```
1 if (32 <= c && c <= 126) {        /* 32 is blank, 126 is tilde */
2 if (' ' <= c && c <= '~') {
3 if (isprint(c)) {                 /* K&R pp. 43, 249; #include <ctype.h>;
```


4 King pp. 527, 614−615 */

There must be exactly two variables. Name them **c** and **b**. You get no credit if you write **getchar** more than once.

You get no credit for Homeworks 5.3 or 5.4 unless you hand in the output of Homework 5.4.

▲

Differences between & and &&

```
expr1 & expr2
expr1 && expr2
```

(1) The value of the expression **expr1 && expr2** is always 1 or 0. It's 1 if **expr1** and **expr2** are both non-zero, and 0 otherwise. But the value of the expression **expr1 & expr2** can be any integer at all, since the **&** operator computes the bits individually.

(2) **&** evaluates **expr1** and **expr2** in an unpredictable order, but **&&** always evaluates **expr1** first.

(3) **&** always evaluates both **expr1** and **expr2**, but **&&** evaluates **expr2** only if **expr1** is not 0. If **expr1** is 0, **&&** will always yield a 0 so there is no point in evaluating **expr2**.

Ditto for **|** and **||**, and **~** and **!**. See K&R pp. 48−49; King pp. 452−453 for the bitwise operators **&**, **^**, **|**, and **~**; K&R pp. 41−42; King pp. 64−65 for the non-bitwise operators **&&**, **||**, and **!**.

□