Character IO, Expressions

Class 14

Character IO

- the simplest IO consists of a reading a single character from the standard input stream with `getchar`
- and writing a single character to the standard output stream with `putchar`
- all data is in chunks of one byte, which is the size of `char`
- a byte in a file could be any value from `0000 0000` to `1111 1111` which in hex is `0x00` to `0xff`
- all 256 values are valid data
- but there must also be the ability to detect end-of-file
- thus the return type of `getchar` is not char but int

Program 1.5.1

```
 1  /*
 2   * echo input to output
 3   * one character at a time
 4   * K&R 1.5.1
 5   */
 6
 7  #include <stdio.h>
 8
 9  int main(void)
10  {
11    int c;
12    c = getchar();
13
14    while (c != EOF)
15    {
16      putchar(c);
17      c = getchar();
18    }
19    return 0;
20  }
```

- 2-space indents
- braces vertically aligned
- main function return type and void argument list
- space after while, no space after main, putchar,getchar
- return from main

- obviously, the argument to putchar is also an int

Program 1.5.1 Version 2

```
1  /*
2   * echo input to output
3   * one character at a time
4   * K&R 1.5.1 version 2
5   */
6
7  #include <stdio.h>
8
9  int main(void)
10 {
11   int c;
12
13   while ((c = getchar()) != EOF)
14   {
15     putchar(c);
16   }
17
18   return 0;
19 }
```

- assignment has a return value
- more succinct
- classic C idiom
- extra parentheses needed for operator precedence = vs !=

Running the Program

- `$ ./prog1_5_1v2` (remember Ctrl-d to end keyboard input)
- `$ echo $?`

- can redirect input
  `$ ./prog1_5_1v2 < prog1_5_v2.c`
- can redirect input and output
  `$ ./prog1_5_1v2 < prog1_5_v2.c > foo`

One More Example

```
1  #include <stdio.h>
2  #define IN 1
3  #define OUT 0
4
5  int main(void)
6  {
7    int c;
8    unsigned lines = 0;
9    unsigned words = 0;
10   unsigned characters = 0;
11   unsigned state = OUT;
12   while ((c = getchar()) != EOF)
13   {
14     ++characters;
15     if (c == '\n')
16     {
17       ++lines;
18     }
19     if (c == ' ' || c == '\n' || c == '\t')
20     {
21       state = OUT;
22     }
23     else if (state == OUT)
24     {
25       state = IN;
26       ++words;
27     }
28   }
29   printf("%u %u %u\n", lines, words, characters);
30   return 0;
31 }
```

- #define directive vs. const
- no semicolons on preprocessor lines
- one variable per declaration

K & R Sections

you should know:

- chapter 2, all
- chapter 3, all except 3.7 and 3.8
  - we NEVER use break
  - continue is so sloppy you should be embarrassed to even consider it
  - we NEVER use goto
- 6.1 through 6.3

K & R Sections

we will cover:

- 2.9
- chapter 4
- chapter 5
- chapter 6
- chapter 7
- pieces of chapter 8 and appendix B

Overflow and Underflow

```
1  /* illustrate wrapping */
2  #include <stdio.h>
3  #include <stdint.h>
4
5  int main(void)
6  {
7    uint8_t uvalue = 0;
8    int8_t svalue = -128;
9
10   printf("start: %u\n", uvalue);
11   uvalue--;
12   printf("minus 1: %u\n", uvalue);
13   uvalue++;
14   printf("plus 1: %u\n\n", uvalue);
15
16   printf("start: %d\n", svalue);
17   svalue--;
18   printf("minus 1: %d\n", svalue);
19   svalue++;
20   printf("plus 1: %d\n", svalue);
21   return 0;
22 }
```

- C does not define behavior — machine dependent
- on Intel-based Unix, everything simply wraps around

```
$ ./wrap
start: 0
minus 1: 255
plus 1: 0

start: -128
minus 1: 127
plus 1: -128
```

Bitwise Operators

- C has six bitwise operators
- integer argument (but we don't use them on char)
- mostly they only make sense with unsigned values

&	bitwise and
|	bitwise or
^	bitwise exclusive or
<<	left shift
>>	right shift
~	unary bitwise complement

and

- in artwork, a stencil is a thin sheet with a cutout pattern
- lay the sheet on a surface and spray with paint
- where the sheet is, paint is blocked
- where the cutouts are, paint is applied to the surface
- the sheet areas of the stencil mask the surface

- bitwise and is typically used as a stencil sheet
- zeroes are solid stencil sheet
- ones are cutouts that let the paint through

and

```
1 uint8_t value;
2 uint8_t mask = 0xa5; /* 1010 1001 */
3
4 /* lets through 4 bits, blocks the others */
5 uint8_t result = value & mask;
```

- value is like the spray paint
- mask is the stencil sheet

```
value ->    1101 1010
mask  ->  & 1010 1001
          -----------
result ->   1000 1000
```

or

```
1 uint8_t value;
2 uint8_t inkpad = 0xa5; /* 1010 1001 */
3
4 /* guarantees 4 bits, allows others */
5 uint8_t result = value | inkpad;
```

- in contrast, or is like an inkpad, or a stencil with ink on it
- value is still like spray paint
- image appears from paint or inked spots (inclusive or)

```
value  ->   1101 1010
inkpad -> | 1010 1001
          -----------
result ->   1111 1011
```

xor

```
1 uint8_t value;
2 uint8_t inkpad = 0xa5; /* 1010 1001 */
3
4 /* allows only one through */
5 uint8_t result = value ^ inkpad;
```

- xor doesn't allow too much of a good thing
- value is spray paint, inkpad is also
- image appears from paint or inkpad but not both

```
value  ->   1101 1010
inkpad -> ^ 1010 1001
           -----------
result ->   0111 0011
```

Left Shift

- left shift is the equivalent of multiplying by a power of 2
- low-order bits are filled with zeroes

```
1 uint8_t value = 0x29; /* 0010 1001 = 41 */
2 value <<= 2; /* now 1010 0100 = 41 x 4 = 164 */
```

```
value ->  0010 1001
 <<2      ---------
result -> 1010 0100
```

- textbook error on page 49, lines 5–6
- says shift amount must be positive
- actually shift amount must be non-negative
- shift can be zero (no change)

Right Shift

- for an unsigned value, strictly equivalent to dividing by a power of 2
- zeroes are shifted into the most significant positions

- for a signed value, machine dependent
- are zeroes or ones shifted into the most significant positions?
- on Intel-based Unix, equivalent to dividing by power of 2
- details later, when we study 2's complement representation

```
int8_t svalue = -8;
svalue >>= 2; /* svalue is now -2 */
```

Example

- from K&R page 49

```
/* get n bits from x starting at position p */
/* lsb is position 0 */
/* p = 6, n = 2 means positions 6 and 5 */
unsigned getbits(unsigned x, unsigned p, unsigned n)
{
  return (x >> (p + 1 - n)) & ~(~0 << n);
}
```