Computer Programming: Skills & Concepts (CP)
Characters

Ajitha Rajan

Monday 30 October 2017

Last lecture

- Practical programming

This lecture

- Robust input handling
- Characters in C

`scanf` – erroneous input

What if the user types a word, when an integer is required?

As already noted in tutorials:

> *Apart from the* action *performed by* `scanf` *(reading, or attempting to read, the object of the specified type),* `scan` returns *an integer, which is the* number of input items assigned*. This may be fewer than provided for, or even zero, in the event of a matching failure.*

This returned value can be used to test for a successful read:

```
scanf("%d", &a) == 1
```

if and only if an integer was successfully read into `a`.

scanf – error-checking our input

Suppose we want to read in an integer to x:
We can *test* for success by saving the returned value of scanf:

```
read_succ = scanf("%d", &x);
if (read_succ == 1) {
   ....
} else {
   ....
}
```

What about the else branch?

- Print an error message and terminate?
- Can give the user a second try.

scanf error-checking – first attempt

```
printf("Please input an integer: ");
read_succ = scanf("%d", &x);
if (read_succ == 1) {
   ....
 }
else {                          /*  read_succ must have been 0 */
   printf("That wasn't an integer! Try again: ");
   read_succ = scanf("%d", &x);
   ....
}
```

PROBLEM: Guaranteed to fail on error ...
WHY?

scanf error-checking – 'skipping over'

`scanf("%*s");` means 'skip over' first item in read-buffer from standard input (the `s` is for 'string' (sequence of non-whitespace characters), the `*` for 'don't save').

```
printf("Please input an integer: ");
read_succ = scanf("%d", &x);
if (read_succ == 1) {
  ....
 }
else {                          /* read_succ must have been 0 */
  scanf("%*s");                /* scan the bad-input, don't save */
  printf("That wasn't an integer! Try again: ");
  read_succ = scanf("%d", &x);
  ....
 }
```

scanf error-checking – loops

```
printf("Please input an integer: ");
read_succ = scanf("%d", &x);
if (read_succ != 1) {     /* read_succ must have been 0 */
  while (read_succ != 1) {
    scanf("%*s");     /* scan bad-input, don't try to save */
    printf("That wasn't an integer! Try again: ");
    read_succ = scanf("%d", &x);
  }
}
....     /* Now we definitely have an int; do the work */
```

Try it with the Fibonacci programs!

Characters

What is it that input handling is *actually* reading from the terminal? Not integers, doubles, or whatever, but *characters*.

The various symbols (`'A'`, `'a'`, `'0'`, `';'`, `'@'`, etc) that you might find on the keyboard, together with control characters such as `'\n'` (newline), all have integer codes (ASCII).

These integers are rather small, so can be wasteful (but sometimes *necessary*) to use a variable of type `int` to represent them.

The `char` type

The type `char` is like a small integer type, just big enough (a *byte*) to hold the usual (in the 1970s) character set.

- *Advantage* of `char` over `int`: saves space. Is the type used in many text-processing encodings.
- *Disadvantage* of `char` over `int`: cannot be used in certain situations (as we'll see).

Oddly enough, `'a'`, `'b'`, `'c'`, etc., denote integer constants and not characters.

Bytes and `char`

A *byte* is a *binary number* of length 8 (8 'bits').

- 2 options for each bit $\Rightarrow$ a byte can take on $2^8 = 256$ possible values (0 up to 255).
- This is enough to cover the English alphabet + other relevant symbols ...

Bytes and `char`

A *byte* is a *binary number* of length 8 (8 'bits').

- 2 options for each bit $\Rightarrow$ a byte can take on $2^8 = 256$ possible values (0 up to 255).
- This is enough to cover the English alphabet + other relevant symbols ...

If you want to play 麻將, listen to Խաչատրյան, discuss the plays of Ἀριστοφάνης, or just ask somebody what their Erdős number is, you need more. In the modern world, real characters have values up to 1114111 – but the C `char` is still 8 bits. If you need to deal with non-ASCII, consult a book or the Web!

Bytes and `char`

A *byte* is a *binary number* of length 8 (8 'bits').

- 2 options for each bit $\Rightarrow$ a byte can take on $2^8 = 256$ possible values (0 up to 255).
- This is enough to cover the English alphabet + other relevant symbols ...

If you want to play 麻將, listen to Խաչատրյան, discuss the plays of Ἀριστοφάνης, or just ask somebody what their Erdős number is, you need more. In the modern world, real characters have values up to 1114111 – but the C `char` is still 8 bits. If you need to deal with non-ASCII, consult a book or the Web!

- The C `char`, like `int` and `float`, is a *signed* type, so actually takes values from -128 to 127. Usually it's better to use `unsigned char`, which really does take values 0 to 255.

Some char values

'a'	97	'b'	98	'z'	112
'A'	65	'B'	66	'Z'	90
'0'	48	'1'	49	'9'	57
'&'	38	'*'	42	'\n'	10
' '	32	'\a'	7	'\r'	13

' ' is the space character.

'\r' is the carriage return character.

'\a' is a special character that rings a bell!

I/O with characters

- `getchar()`: returns the next character from the `input stream` (could be characters typed at a keyboard, or read from a file). If the end of the stream has been reached (user types CTRL/D or the end of the file is reached) the special value EOF (which is -1 on most systems, but always refer to it as `EOF`) is returned.
- `putchar(c)`: writes the character `c` to the *output stream* (could be the screen, or another file).

These functions are included in `<stdio.h>`.
NOTE: `getchar()` returns an `int`, not a `char` ! This is so that it can return all the possible `unsigned chars` as well as the value EOF.

Library functions

In addition, `#include <ctype.h>` gives us various functions on characters:

- `isalpha(c)`: is `c` alphabetic?
- `isupper(c)`: is `c` upper case?
- `isdigit(c)`: is `c` a digit (`0` to `9`)?
- `toupper(c)`: if `c` is a lower case letter, return the corresponding upper case letter; otherwise return `c`.

. . . and several others: see Kelley and Pohl A.2, or the `isalpha` man-page.

Printing Roman numerals

```
void PrintNum(int n) {
  while (n > 0) {
    if (n >= 100) {
      n = n - 100;  putchar('C');
    } else if (n >= 90) {
      n = n + 10;  putchar('X');
    } else if (n >= 50) {
      n = n - 50;  putchar('L');
    } else if (n >= 40) {
      n = n + 10;  putchar('X');
    } else if (n >= 10) {
      n = n - 10;  putchar('X');
```

```
    } else if (n >= 9) {
      n = n + 1;  putchar('I');
    } else if (n >= 5) {
      n = n - 5;  putchar('V');
    } else if (n >= 4) {
      n = n + 1;  putchar('I');
    } else {
      n = n - 1;  putchar('I');
    }
  }
}
```

Idiom for single character I/O

We can do a surprising amount by filling in the following template:

```
int c;

while ((c = getchar()) != EOF) {
  /*  Code for processing the character c. */
}
```

The `while`-loop condition is a bit tricky: it reads a character from the input, assigns it to `c` *and* tests whether the character is `EOF` (i.e., whether we have reached the end of the input)!

Continuing the Roman theme: Caesar cypher

```
const int OFFSET = 13, NUMLETS = 26;

int c, ord;   /* Why is c declared as int and not char? */

while ((c = getchar()) != EOF) {
  c = toupper(c);
  if (isupper(c)) {
    ord = c - 'A';                    /* Integer in range [0,25] *
    ord = (ord + OFFSET) % NUMLETS; /* permute by offset */
    c = ord + 'A';                    /* back to char */
  }
  putchar(c);
}
```

Example: Letter frequencies

```
#define NUMLETS 26
int c,  i, count[NUMLETS];

for (i = 0; i < NUMLETS; i++) count[i] = 0;
while ((c = getchar()) != EOF) {
  c = toupper(c);
  if (isupper(c)) {
    i = c - 'A';                    /* Integer in [0,25] */
    count[i]++;
  }
}
for (i = 0; i < NUMLETS; i++) {
  printf("%c: %d\n", i + 'A', count[i]);
}
```

Idiom for line-oriented I/O

We can do a surprising amount by filling in the following template:

```
int c;

while ((c = getchar()) != EOF) {
  if (c == '\n') {
    /* Code for processing the line just read. */
  } else {
    /* Code for processing the character c. */
  }
}
```

Example: recording line lengths

```
int c, charCount = 0, lineCount = 0;

while ((c = getchar()) != EOF) {
  if (c == '\n') {
    lineCount++;
    printf("  [Line %d has %d characters]\n",
                              lineCount, charCount);
    charCount = 0;
  } else {
    charCount++;
    putchar(c);
  }
}
```

Input and output redirection

Suppose we have compiled a program, similar to the ones considered earlier, and placed the resulting object code in the file `prog` (*maybe done by creating a* `Makefile` *and using* `make`; *or alternatively just by copying* `a.out` *into* `prog`).

By default, input is from the keyboard, and output is to the screen. So

- Typing `./prog` in the shell window runs `prog`, with input being taken from the keyboard, and output being written to the shell window.

However, by extending the command, we may redirect input from the keyboard to a nominated input file, and redirect the output from the screen to a nominated output file.

- `./prog < data` takes input from the file `data`, but continues to send output to the shell window.
- `./prog > results` takes input from the keyboard, but sends output to the file `results`.
- `./prog < data > results` takes input from the file `data`, and sends output to the file `results`.

Reading material :)

Kelley and Pohl, subsections 3.2, 3.3 and 3.9