CSE 2421

The C Language – Part 2 – I/O

Character value encodings

- Linux/Unix systems use the ASCII character encoding format
- Most IBM products use the EBCDIC character encoding format
- There is also a character encoding format called Unicode
- For this class we will only work with ASCII
- If you are interested in this type of thing, I suggest https://en.wikipedia.org/wiki/Character_encoding

as a short tutorial/history lesson.

C functions and I/O

- **Reminder:** Before a function in a C program is invoked, or called, it must be declared. Otherwise, the compiler will output an error message when you compile. Remember also that a definition of a function includes a declaration.
- When we want to use I/O functions in the C standard library, we will include stdio.h in our source file, which contains declarations of these library functions, so that the compiler can do type-checking of our calls to these functions.
- We will only look at some of the most commonly used of these functions here.
  - They are all you need for the labs in this course (and all you are expected to use).

Basic I/O

- There is no input or output defined in C itself; functions have been written and incorporated into the standard library for input and output (stdio).
- Two basic types of I/O:
  - Character based (no format specifiers) – *character by character* I/O
    - getchar() - input
    - putchar(c) - output
  - Formatted - zero or more characters, with a requirement to specify the format of the input or output (how the sequence of characters read or written is to be interpreted):
    - scanf(parameters go in here) - input - white space is important!!!
    - printf(parameters go in here) - output
    - These functions use *format strings* to specify interpretation (% before string)
- Example

```
#include <stdio.h>
int main()
{     /* check1.c - shows calls to scanf() and printf() */
      int x;
      scanf("%d\n", &x); /* Notice address operator & */
      printf("x=%d\n", x);
}
```

getchar() and putchar()

- These functions can be used to input and output one character at a time.
- Declarations (interfaces):
  - int getchar(void);
  - int putchar(int);   /* the int value returned can be ignored */
- Notice that both of these functions use the **integer** values which correspond to a given ASCII character. The integer must be cast before being assigned to a char variable, or must be printed with a %c format code to be output as a char.
- An integer is returned, rather than a char, because certain text file control values (EOF, or end of file, for example) do not have one byte ASCII encodings.
- Now would be a good time to review the ASCII character chart on piazza ☺ Take note of the values for ‘0’–’9’, ‘a’–’z’, ‘A’–’Z’ and their relationship.

Example with getchar()

```
/* assume #include <stdio.h> */
…
int code;
…
printf(“Please enter the appropriate single letter code, followed by enter: \n”);
code = getchar();
printf(“The code entered was: %c \n”, code); /* outputs int as a char */
-----------------------------------------------------------------
/* another way to do it – again, assume #include <stdio.h> */
…
char code;
…
printf(“Please enter the appropriate single letter code, followed by enter: \n”);
code = (char) getchar();    /* cast int value read from input to char */
printf(“The code entered was: %c \n”, code);
```

What happens to EOF in the 2nd example???

Formatted I/O: printf vs scanf

- Both format I/O
- Both manipulate "standard I/O" location – the keyboard for input, and the terminal display for output (although we can use something called redirection to change this – more below)
- printf - output
  - Converts values to a character form, according to the format string, and prints them to "standard out" (stdout)
- scanf - input
  - Converts characters from "standard in" (stdin), according to the format string, and followed by **pointer** arguments (i.e., addresses), indicating where the resulting values are to be stored

ANSI C Formatted I/O

- Input:
  - scanf(“format specification string”, &param1, &param2, &param3, …);
  - A format specification string is a string literal defining the format from which input should be read
  - scanf can have an indeterminate number of parameters. The 2nd through the nth parameters are the addresses of variables into which values are placed (passing a reference)
  - There is a % format substring within the first parameter that represents the format to be used for each of the next n-1 parameters
- Output:
  - printf(“format specification string”, param1, param2, param3, …);
  - A format specification string is a string literal defining the format in which output should be formatted
  - printf can have an indeterminate number of parameters. The 2nd through the nth parameters are values to be printed (pass by value)
  - There is a % format substring within the first parameter that represents the format to be used for each of the next n-1 parameters

Format Strings

When programming in C, you use **format strings** — the percentage sign and a conversion code, for the most part (although some other characters between % and the conversion character are optional – see below) — as placeholders for variables you want to read from input or display. The above table shows the format strings and what they display...

Conversion Character	Displays Argument (Variable's Contents) As
%c	Single character
%d, %i	Signed decimal integer (int)
%e	Signed floating-point value in E notation
%f	Signed floating-point value (float)
%g	Signed value in %e or %f format, whichever is shorter
%o	Unsigned octal (base 8) integer (int)
%s	String of text (NULL terminated)
%u	Unsigned decimal integer (int)
%x	Unsigned hexadecimal (base 16) integer (int)
%%	(percent character)

Codes for Formatted I/O: "%-+ 0*w*.*pmc*"

`-`	left justify
`+`	print with sign
space	print space if no sign
`0`	pad with leading zeros
w	min field width
p	precision
m	conversion character:
	`h` short, `l` long, `L` long double
c	conversion character:

`d,i`	integer	`u`	unsigned
`c`	single char	`s`	char string
`f`	double (`printf`)	`e,E`	exponential
`f`	float (`scanf`)	`lf`	double (`scanf`)
`o`	octal	`x,X`	hexadecimal
`p`	pointer	`n`	number of chars written
`g,G`	same as `f` or `e,E` depending on exponent		

Formatted output – printf examples

```
/* NOTE: The codes for formatted I/O from above are used */
```

```
printf("%d\t%d\n", fahr, celsius);
```

Causes the values of the two integers fahr and celsius to be printed, with a tab (\t) between them, and followed by a newline.

```
printf("%3d\t%6d\n", fahr, celsius);
```

To print the first number to three digits wide, and the second to six digits wide

NOTE: Each % construction in the first argument of printf is paired with the corresponding second argument, third argument, etc.; they must match up properly by number and type, or you will get wrong answers or a compile time error.

```
printf("\na=%f\nb=%f\nc=%f\nd=%f",a,b,c,d);
What does this statement print?
```

printf examples

```
float x = 3.14159;
printf(“output:”);
printf("%4.3f\n", x);
printf("%4f\n", x);
printf("%.3f\n", x);
```

```
output:
3.142
3.141590
3.142
```

- *minimum* field width is 4! Notice it rounds up.
- *minimum* field width is 4, but no precision specified – 6 digits is default
- Explicitly says 3 numerals after decimal point

printf example

```
int y = 25;
printf("%d\n", y);
printf("%i\n", y);
printf("%4d\n", y);
printf("%1d\n", y);
printf("%05d\n", y);
```

output:

```
25
25
  25
25
00025
```

- two spaces before two-digit value
- minimum field width specified is too small to print the value of the data, so printf ignores it!
- leading zeroes used to fill minimum field width

scanf examples

```
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year); /* call by value! */
Input:
01/29/13
Month has the value 1
Day has the value 29
Year has the value 13
Input:
111/222/2018
Month has the value 111
Day has the value 222
Year has the value 2018

int anInt;
scanf("%i%%", &anInt);
Input:
23%
anInt has the value 23
Input:
152%
anInt has the value 152
```

More scanf examples

```
int int_1; long long_1;
scanf("%d  %ld", &int_1, &long_1);
Input:
-23 200
int_1 has the value -23    /*stored as 4 bytes */
                           /*on stdlinux*/
long_1 has the value 200   /*stored as 8 bytes */
                           /*on stdlinux*/
double d;
scanf("%lf", &d);
Input:
3.14
d has the value 3.14       /*stored as 8 bytes */
                           /*on stdlinux*/
```

More scanf examples

- IMPORTANT: scanf *ignores* ***leading*** (but **not** following) white space characters when it reads ***numeric*** values from input:

```
    int i;
    printf("Enter an integer:\n");
    scanf("%d", &i);
```

/*scanf will ignore any leading white space characters */

More scanf examples

```
char string1[10]; /* IMPORTANT: string must hold 9 chars + null */
scanf("%9s", string1); /*array identifier is the address of 1st element*/

Input:
VeryLongString
string1==“VeryLongS”

int int1;
scanf("%*s %i", &int1);    /*read arbitrary length*/
                          /*string of non-numeric chars before int1 */
Input:
Age: 29
int1 has the value 29

NOTE: pressing the enter key causes characters that are being held in the
  input buffer to be transferred to standard in (stdin)
```

Still more scanf examples

```
int int1, int2;
scanf("%2i", &int1);
scanf("%2i", &int2);
Input:
2345
Then:
int1 has the value 23
int2 has the value 45
```

NOTE: pressing the enter key causes characters being held in the input buffer to be transferred to standard in (stdin)

Scanf format examples

Letter	Type of Matching Argument	Auto-skip; Leading White-Space	Example	Sample Matching Input

%	% (a literal, matched but not converted or assigned)	no	int anInt;
			scanf("%i%%", &anInt);
				23%

d	int – in decimal format	yes	int anInt; long l;
			scanf("%d %ld", &anInt, &l);
				-23 200

i	int – in decimal, octal, or hex format	yes	int anInt;
			scanf("%i", &anInt);
				15 017 0xF

o	unsigned int	yes	unsigned int aUInt;
			scanf("%o", &aUInt);
				023

u	unsigned int	yes	unsigned int aUInt;
			scanf("%u", &aUInt);
				23

x	unsigned int	yes	unsigned int aUInt;
			scanf("%x", &aUInt);
				1A

a, e, f, g	float or double	yes	float f; double d;
			scanf("%f %lf", &f, &d);
				1.2 3.4

c	char	no	char ch;
			scanf(" %c", &ch);
				Q

s	array of char	yes	char s[30];
			scanf("%29s", s);
				hello

n	int	no	int x, cnt;
			scanf("X: %d%n", &x, &cnt);
				X: 123 (cnt==6)

[	array of char	no	char s1[64], s2[64];
			scanf(" %[^\n]", s1);
			scanf("%[^\t] %[^\t]", s1, s2);
				Hello World
				field1 field2

scanf – dealing with size differences

- double: use %lf
  - l is for “long” since a double is longer than a float
- long double: use %Lf
  - L (upper case) because it is really long
- short: use %hd
  - h is for “half” since a short is smaller than an int
- long: use %ld
  - l is for “long” since a long is longer than an int

printf – dealing with sizes

- long: use %ld
  - Only matters for numbers that are too big for an int

- long double: use %Lf

printf promotes float parameters to double, so %f works for both float and double

printf promotes smaller integer type parameters to int, so %d works for int and smaller integer types

Pointers and printf

- %p is for printing a pointer value in hex
- When we get to pointers, don’t forget we can print them
- If we can print them, we may be able to debug them
- Unlike other languages we can actually find out where things are in memory this way

Redirection of stdin, stdout and stderr

- In Unix/Linux, there are 3 file descriptors that are explicitly defined to represent 3 distinct data streams: *stdin* (0), *stdout* (1) and *stderr*(2).
- IMPORTANT: By default, in Unix/Linux, C programs read input from "standard input (*stdin*)" (usually, the keyboard) and write output to "standard output (*stdout*)" (usually, the screen). Unix/Linux error messages go to "standard error" (*stderr*) (also the screen).
- We can change the destination of each of these data streams from the Unix/Linux command line using what is called redirection:
  - < redirects the input, and
  - > redirects the output,
  - 2> redirects errors
- You can redirect just the input or just the output or just errors (that is, you do not have to redirect all of them, but you could).
- For labs, from time to time it will be convenient to redirect input and allow output to go to standard out (the screen). Obviously, there shouldn't be any Unix/Linux errors to redirect. ☺

Redirecting Standard input

- It is important to make sure that the input file has an end of file (EOF) at the end. So, when you are creating/editing your input file, be sure you hit *enter* at the end of the last line of the file.
- Example:
  - prog1 < prog1.input

- This example runs program, ***prog1***, and, rather than looking for input from the keyboard, gets its input from the file ***prog1.input***
- The file ***prog1.input*** is expected to be in the same directory in which ***prog1*** is executing. The program will not look for the file anywhere else ***unless*** a full pathname is specified for the input file.
- If you redirect ***stdin***, the program will expect ***all*** input that it requires (until the program terminates) to come from the specified file (i.e. you can't put part of the input to the program in a file and then when the data in the file runs out, input the rest of it from the command line).
- Any output to ***stdout*** or any Unix/Linux errors messages to ***stderr*** would go to the screen.

Redirecting standard output

- Example 1:

```
% prog1 > output.file
```

- This will run *prog1* and write output to *output.file*
  - If *output.file* does not exist, it will create one.
  - If *output.file* does exist, any data currently in the file will be erased and only data from the most current run of *prog1* will be in the file.
  - If you do not use a full pathname for the output file, it will only look for the file in the current directory
- Example 2:

```
% prog1  >> output.file
```

- This will run *prog1* and write output to *output.file*
  - If *output.file* does not exist, it will create one.
  - If *output.file* does exist, output from the current run of the program will be *appended to the end of the file*.
  - If you use this option, be very careful not confuse yourself with respect to what output came from which execution of your program.
- Both of these examples would expect input to come from the keyboard and any Unix/Linux errors would be sent to the screen

Redirecting standard stderr

- In most instances, it's not a good idea to redirect ***stderr***, but from time to time…
- Example 1:

```
% prog1  2> error.file
```

- This will run ***prog1*** and write any Unix/Linux error messages to ***error.file***
  - If ***error.file*** does not exist, it will create one.
  - If ***error.file*** does exist, any data currently in the file will be erased and only error messages from the most current run of ***prog1*** will be in the file.
  - If you do not use a full pathname for the error file, it will look for the file in the current directory
- Example 2:

```
% prog1   2>> error.file
```

- This will run ***prog1*** and write any Unix/Linux errors to ***error.file***
  - If ***error.file*** does not exist, it will create one.
  - If ***error.file*** does exist, output from the current run of the program will be *appended to the end of the file*.
  - If you use this option, be very careful not confuse yourself with respect to what errors came from which execution of your program. It would be sad to find out that you stayed up all night looking for an error you see in error.file that you fixed before midnight.
- Both of these examples would expect input to come from the keyboard and ***stdout*** would be sent to the screen

Combinations

- What if I want to redirect ***stdout*** and ***stderr*** to the same file?
  - prog1 > stdout.file 2>&1

- Do
  - prog1 > stdout_stderr.file 2>&1

    and
  - prog1 2>&1 >stdout_stderr.file

  Do the same thing??

- No! Order matters!
  - prog1 > /home/user/stdout_stderr.file 2>&1

    This will send both ***stdout*** and ***stderr*** to stdout_stderr.file

  - prog1 2>&1 > stdout_stderr.file

    This will send ***stdout*** to stdout_stderr.file, and ***stderr*** to what was previously ***stdout***.

- When you perform a shell redirection, the left side of the redirection goes to where the right side of the redirection currently goes. Meaning in 2>&1, it sends ***stderr*** (2) to wherever ***stdout*** (1) currently goes.

  But if you, afterwards, redirect ***stdout*** somewhere else, ***stderr*** doesn't go with it. It continues to go wherever ***stdout*** was previously going. This is why in the first example, both ***stdout*** and ***stderr*** will go to the same place, but in the second they won't.