

CMSC 313
COMPUTER ORGANIZATION
&
ASSEMBLY LANGUAGE
PROGRAMMING

LECTURE 11, SPRING 2013



TOPICS TODAY

- **C Input/Output**
- **Characters & Strings in C**
- **Structures in C**
- **Project 4**



C INPUT/OUTPUT



stdin, stdout, stderr

C opens three input/output devices automatically:

stdin

The “standard input” device, usually your keyboard

stdout

The “standard output” device, usually your monitor

stderr

The “standard error” device, usually your monitor

Some C library I/O functions automatically use these devices

Formatted Console Output

- `printf()` outputs formatted text to `stdout`

```
printf( format, arg1, arg2, ... );
```

- **Example:**

```
int n = 3 ;  
printf ( "Value = %d\n", n) ;
```

- **format is a string containing**
 - conversion specifications
 - literals to be printed

printf() conversions

Conversions specifications begin with % and end with a conversion character.

Between the % and the conversion character MAY be, in order

A minus sign specifying left-justification

The minimum field width

A period separating the field width and precision

The precision that specifies

Maximum characters for a string

Number of digits after the decimal for a floating point

Minimum number of digits for an integer

An h for “short” or an l (letter ell) for long

man printf for more documentation.

Common printf() Conversions

- %d** print integer as a decimal number (base 10)
- %u** print integer as unsigned number
- %s** print string
- %f** print double as a floating point number
- %x** print integer in hexadecimal (base 16)
- %c** print integer as ASCII character
- %p** print pointer in hexadecimal (implementation dependent)

printf() Examples

```
int anInt = 5678;
double aDouble = 4.123;
#define NAME "Bob"

/* what is the output from each printf( ) */
printf ("%d is a large number\n", anInt);
printf ("%8d is a large number\n", anInt);
printf ("% -8d is a large number\n", anInt);
printf ("%10.2f is a double\n", aDouble);
printf( "The sum of %d and %8.4f is %12.2f\n",
        anInt, aDouble, anInt + aDouble);
printf ("Hello %s\n", NAME);
```


Formatted Output Example

Use field widths to align output in columns

```
int i;
for (i = 1 ; i < 5; i++)
    printf("%2d %10.6f %20.15f\n", i, sqrt(i), sqrt(i));
```

```
12 1234567890 12345678901234567890
1   1.000000   1.000000000000000000
2   1.414214   1.414213562373095
3   1.732051   1.732050807568877
4   2.000000   2.000000000000000000
```

Keyboard Input

- `scanf` reads user input from `stdin`.
- Syntax for `scanf()` is similar to `printf()`
`scanf(format, arg1, arg2, ...)`
- The format string similar structure to `printf()`.
- The arguments must be *addresses* of the variables.

scanf() format string

The `scanf()` format string usually contains conversion specifications that tell `scanf()` how to interpret the next “input field”. An input field is a string of non-whitespace characters.

The format string usually contains

- Blanks or tabs which are ignored

- Ordinary characters which are expected to match the next (non-whitespace) character input by the user

- Conversion specifications usually consisting

 - % character indicating the beginning of the conversion

 - An optional h, l (ell) or L

 - A conversion character which indicates how the input field is to be interpreted.

Common scanf() conversions

- %d** a decimal (integer) number
- %u** an unsigned decimal (integer) number
- %x** a hexadecimal number
- %f** a floating point number with optional sign, decimal point, and exponent
- %s** a string delimited by white space, NOT an entire line
- %c** a single character (possibly a whitespace char)

scanf() examples

```
int age;  
double gpa;  
char initial;  
  
printf(" input your middle initial: ");  
scanf ("%c", &initial );           // note &  
printf("Input your age: ");  
scanf( "%d", &age );  
printf(" input your gpa: ");  
scanf ("%lf", &gpa );
```

Unix I/O redirection

- **Redirect input** (read from `infile` instead of keyboard):
`a.out < infile`
- **Redirect output** (write to `outfile` instead of screen):
`a.out > outfile`
- **Redirect both**:
`a.out < infile > outfile`
- **Redirect `stdout` and `stderr` to `outfile`**
`a.out >& outfile`
- **Redirect `stdout` to `outfile` and `stderr` to `errfile`**
`(a.out > outfile) >& errfile`

Text File I/O

- Use `fprintf()` and `fscanf()` functions instead of `printf()` and `scanf()`.
- Must open file before reading/writing: `fopen()`
- Must close file after all done: `fclose()`
- Use file handle to specify file.
- File handle returned by `fopen()`:

```
FILE *myFile ;
myFile = fopen ("bob.txt", "r") ;
if (myFile == NULL) {
    /* handle the error */
}
```

fopen()

`fopen()` requires two parameters

1. The name of the text file to be opened

2. The text file open “mode”

“r” open the file for reading only

“w” create the file for writing; delete existing file

“a” append; open or create the file for writing at the end

“r+” open the file for reading and writing

“w+” create the file for reading & writing; deletes existing file

“a+” open or create the file for reading or writing at the end

fscanf.c

```
#include <stdio.h>
#include <stdlib.h> /* for "exit" */
int main ( )
{
    double x ;
    FILE *ifp ;

    /* try to open the file for reading, check if successful */
    /* if it wasn't opened exit gracefully */
    ifp = fopen("test_data.dat", "r") ;
    if (ifp == NULL) {
        printf ("Error opening test_data.dat\n");
        exit (-1);
    }
    fscanf(ifp, "%lf", &x) ; /* read one double from the file */
    fclose(ifp); /* close the file when finished */

    /* check to see what you read */
    printf("x = %.2f\n", x) ;
    return 0;
}
```

Detecting end-of-file with fscanf

When reading an unknown number of data elements from a file using `fscanf()`, we need a way to determine when the file has no more data to read, i.e, we have reached the “end of file”.

Fortunately, the return value from `fscanf()` holds the key. `fscanf()` returns an integer which is the number of data elements read from the file. If end-of-file is detected the integer return value is the special value `EOF`

EOF example code

```
/* code snippet that reads an undetermined number of integer
   student ages from a file and prints them out as an example
   of detecting EOF
*/
FILE *inFile;
int age;

inFile = fopen( "myfile", "r" );
if (inFile == NULL) {
    printf ("Error opening myFile\n");
    exit (-1);
}

while ( fscanf(inFile, "%d", &age ) != EOF ) {
    printf( "%d\n", age );
}

fclose( inFile );
```

fprintf.c

```
#include <stdio.h>
#include <stdlib.h> /* exit */
int main ( )
{
    double pi = 3.14159 ;
    FILE *ofp ;

    /* try to open the file for writing, check if successful */
    ofp = fopen("test.out", "w") ;
    if (ofp == NULL) {
        printf ("Error opening test.out\n");
        exit (-1);
    }

    /* write to the file using printf formats */
    fprintf(ofp, "Hello World\n");
    fprintf(ofp, "PI is defined as %6.5lf\n", pi);

    fclose(ofp);          /* close the file when finished reading */

    return 0;
}
```

Adapted from Dennis Frey CMSC 313 Fall 2011

CHARACTERS & STRINGS



char type

C supports the `char` data type for storing a single character.

`char` uses one byte of memory.

`char` constants are enclosed in single quotes

```
char myGrade = 'A';  
char yourGrade = '?';
```

ASCII Character Chart

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Special Characters

Use `\` for escape sequences.

For example

`\n` is the newline character

`\t` is the tab character

`\"` is the double quote (necessary since double quotes are used to enclose strings)

`'` is the single quote (necessary since single quotes are used to enclose chars)

`\\` is the backslash (necessary since `\` now has special meaning)

`\a` is beep which is unprintable

Special Char Example Code

What is the output from these statements?

```
printf("\t\tMove over\n\nWorld, here I come\n");
```

Move over

World, here I come

```
printf("I've written \"Hello World\"\n\t many times\n\a");
```

I've written "Hello World"
many times <beep>

Character Library Functions

int isdigit (int c);

Determine if c is a decimal digit ('0' - '9')

int isxdigit(int c);

Determines if c is a hexadecimal digit ('0' - '9', 'a' - 'f', or 'A' - 'F')

int isalpha (int c);

Determines if c is an alphabetic character ('a' - 'z' or 'A' - 'Z')

int isspace (int c);

Determines if c is a whitespace character (space, tab, etc)

int isprint (int c);

Determines if c is a printable character

int tolower (int c);

int toupper (int c);

Returns c changed to lower- or upper-case respectively, if possible

Character Library Functions

Include header file use character library functions:

```
#include <ctype.h>
```

Technically functions take an `int` parameter, not `char`.

Return type is also `int` . `0 = False`, not `0 = True`.

`man ctype.h` for more functions and complete documentation.

Character Input/Output

Use `%c` in `printf()` and `fprintf()` to output a single character.

```
char yourGrade = 'A';  
printf( "Your grade is %c\n", yourGrade);
```

Input char(s) using `%c` with `scanf()` or `fscanf()`

```
char grade, scores[3];
```

`%c` inputs the next character, which may be whitespace

```
scanf("%c", &grade);
```

`%nc` inputs the next n characters, which may include whitespace.

```
scanf( "%3c", scores);    // note -- no & needed
```

Strings in C

- **String = null terminated array of char.**
- **null = '\0'**
- **String constants in double quotes are null terminated.**
- **Strings do not "know" their own length.**
- **Initialization:**

```
char name4[ 20 ] = { 'B', 'o', 'b', 'b', 'y', '\0' };
```

```
char name5[6] = "Bobby"; // NOT assignment, needs 6 slots
```

```
char name6[ ] = "Bobby";
```

String Output

```
char name[ ] = "Bobby Smith";  
printf( "My name is %s\n", name);
```

```
// Right and left justify
```

```
printf ("My favorite books are %12s and %12s\n", book1, book2);  
printf ("My favorite books are %-12s and %-12s\n", book1,  
        book2);
```

Dangerous String Input

```
char name[22];  
printf(" Enter your name: ");  
scanf("%s", name);
```

Why is this dangerous?

Long name will overwrite memory.

Safer String Input

```
char name[ 22 ];  
printf( "Enter your name: ");  
  
scanf("%21s", name);    // note 21, not 22, 1 byte for '\0'
```


C String Library

C provides a library of string functions.

To use the string functions, include `<string.h>`.

Some of the more common functions are listed here on the next slides.

To see all the string functions, type
`man string.h` at the unix prompt.

C String Library (2)

Must #include <string.h>

strlen(const char string[])

Returns length of string, not counting '\0'

strcpy(char s1[], const char s2[])

Copies s2 on top of s1. **Must have enough space in s1 !!!**

The order of the parameters mimics the assignment operator

strcmp (const char s1[] , const char s2[])

Returns < 0, 0, > 0 if s1 < s2, s1 == s2 or s1 > s2 lexicographically

strcat(char s1[] , const char s2[])

Appends (concatenates) s2 to s1. **Must have enough space in s1 !!!**

C String Library (3)

Some safer functions from the C string library:

strncpy(char s1[], const char s2[], int n)

Copies at most n characters of s2 on top of s1.

Does not null terminate s1 if length of s2 >= n !!!

The order of the parameters mimics the assignment operator

strncmp (const char s1[], const char s2[], int n)

Compares up to n characters of s1 with s2

Returns < 0, 0, > 0 if s1 < s2, s1 == s2 or s1 > s2 lexicographically

strncat(char s1[], const char s2[], int n)

Appends at most n characters of s2 to s1.

String Code

```
char first[10] = "bobby";
char last[15] = "smith";
char name[30];
char you[ ] = "bobo";

strcpy( name, first );
strcat( name, last );
printf( "%d, %s\n", strlen(name), name );

strncpy( name, last, 2 );
printf( "%d, %s\n", strlen(name), name );

int result = strcmp( you, first );
result = strncmp( you, first, 3 );

strcat( first, last );
```

Simple Encryption

```
char c, msg[] = "this is a secret message";
int i = 0;
char code[26] = /* Initialize our encryption code */
    {'t','f','h','x','q','j','e','m','u','p','i','d','c',
    'k','v','b','a','o','l','r','z','w','g','n','s','y'} ;

printf ("Original phrase: %s\n", msg);

/* Encrypt */
while( msg[i] != '\0' ){
    if( isalpha( msg[ i ] ) ) {
        c = tolower( msg[ i ] ) ;
        msg[ i ] = code[ c - 'a' ] ;
    }
    ++i;
}
printf("Encrypted: %s\n", msg ) ;
```

Arrays of Strings

An initialized array of string constants

```
char months[][4] = {"Jan", "Feb", "Mar", "Apr",  
                  "May", "Jun", "Jul", "Aug",  
                  "Sep", "Oct", "Nov", "Dec" } ;  
  
int m;  
for ( m = 0; m < 12; m++ )  
    printf( "%s\n", months[ m ] );
```

Alternative: use typedef

```
typedef char Acronym[4] ;  
Acronym months[] = {"Jan", "Feb", "Mar", "Apr",  
                  "May", "Jun", "Jul", "Aug",  
                  "Sep", "Oct", "Nov", "Dec" } ;
```

sprintf()

sprintf() works just like **printf()** or **fprintf()**, but puts its “output” into the specified character array.

The character array must be big enough.

```
char message[ 100 ];  
int myAge = 4;  
  
sprintf( message, “I am %d years old\n”, age);  
printf( “%s\n”, message);
```

STRUCT



Java vs C

- **Suppose you were assigned a write an application about points and straight lines in a coordinate plane.**
- **In Java, you'd correctly design a Point class and a Line class using composition.**
- **What about in C?**

No Classes in C

- ~~Because C is not an OOP language, there is no way to combine data and code into a single entity.~~
- Related data and functions are form an "Abstract Data Type." Accessibility is enforced by a programmer's good judgment and not by the compiler.
- C does allow us to combine related data into a structure using the keyword `struct`.
- All data in a `struct` variable can be accessed by any code.
- Think of a `struct` as an OOP class in which all data members are public, and which has no methods.

Struct definition

```
struct tag
{
    member1_declaration;
    member2_declaration;
    member3_declaration;
    . . .
    memberN_declaration;
};
```

semi-colon !!!



struct is the keyword

tag names this kind of **struct**,

member_declarations are variable declarations which define
the data members.

C struct Example

- Defining a `struct` to represent a point in a coordinate plane

```
struct point ← point is the struct tag
{
    int x; /* x-coordinate */
    int y; /* y-coordinate */
};
```

- Given the declarations

```
struct point p1;
struct point p2;
```

- we can access the members of these `struct` variables:
 - the x-coordinate of `p1` is `p1.x`
 - the y-coordinate of `p1` is `p1.y`
 - the x-coordinate of `p2` is `p2.x`
 - the y-coordinate of `p2` is `p2.y`

Using struct members

```
int main ( )
{
    struct point leftEndPt, rightEndPt, newEndPt;

    printf("Left end point coordinates ");
    scanf( "%d %d", &leftEndPt.x, &leftEndPt.y);

    printf("Right end point's x-coordinate: ");
    scanf( "%d %d", &rightEndPt.x, &rightEndPt.y);

    // add the endpoints
    newEndPt.x = leftEndPt.x + rightEndPt.x;
    newEndPt.y = leftEndPt.y + rightEndPt.y;

    // print new end point
    printf("New endpoint (%2d, %2d)", newEndPt.x, newEndPt.);

    return 0;
}
```

Initializing a struct

```
struct point middle = { 6, -3 };
```

is equivalent to

```
struct point middle ;  
middle.x = 6 ;  
middle.y = -3 ;
```

struct Variants

```
struct point {  
    int x, y;  
} endpoint, upperLeft ;
```

defines the structure named `point`

AND

the variables `endpoint` and `upperLeft` to be of this type.

struct + typedef

```
typedef struct point {  
    int x, y;  
} POINT;
```

POINT is now a TYPE.

```
POINT endpoint ;
```

is equivalent to

```
struct point endpoint;
```


struct assignment

```
struct point p1;  
struct point p2;  
  
p1.x = 42;  
p1.y = 59;  
  
p2 = p1; /* structure assignment copies members */
```

struct within a struct

```
typedef struct line
{
    POINT leftEndPoint;
    POINT rightEndPoint;
} LINE;

LINE line1, line2;

line1.leftEndPoint.x = 3 ;
line1.leftEndPoint.y = 4 ;
```

Arrays of struct

```
LINE lines[5];    // or struct line lines[5];  
  
printf("%d\n", lines[2].leftEndPoint.x);
```

Arrays within a struct

- Structs may contain arrays as well as primitive types

```
struct month
{
    int nrDays;
    char name[ 3 + 1 ];
};
```

```
struct month january = { 31, "JAN"};
```

A bit more complex

```
struct month allMonths[ 12 ] = {
    {31, "JAN"}, {28, "FEB"}, {31, "MAR"},
    {30, "APR"}, {31, "MAY"}, {30, "JUN"},
    {31, "JUL"}, {31, "AUG"}, {30, "SEP"},
    {31, "OCT"}, {30, "NOV"}, {31, "DEC"}
};

// write the code to print the data for September
printf( "%s has %d days\n",
        allMonths[8].name, allMonths[8].nrDays);

// what is the value of allMonths[3].name[1]
```

Size of a struct

As with primitive types, we can use `sizeof()` to determine the number of bytes in a struct

```
int pointSize = sizeof( POINT );  
int lineSize = sizeof (struct line);
```

As we'll see later, the answers may surprise you!

NEXT TIME

- **Parameter passing**
- **Separate Compilation**
- **Scope & Lifetime**

