Daily Course Notes for CS240
Computer Architecture 1
Patrick E. O'Neil

Class 1.
Hand out: Syllabus, UNIX Guide, SHOW Texts
Hand out: Handin -- Taken Prereq? Phone #?
Hand out: Notes. NOTE: DON'T TRUST DUE DATES, EXAM DAYS in notes. Depend on class announcements. I will pass out notes or you can download notes from class home: **http://www.cs.umb.edu/cs240**
BUT All Assignments will be posted in UNIX Directory: ~poneil/cs240
Bring $5/$10. for notes I will hand out in class.

Go over Syllabus. Grading. First Professional programming course: C, UNIX. Can get low-level job as programmer after you learn CS240 properly. Go over texts, when used. My name, office (Draw Map on board), and home phone: call with Qs.

Get UNIX Guide On-Line. First thing to do is Apply. **YOU MUST DO THIS** EVEN IF YOU ALREADY HAVE A UNIX ACCOUNT IN ORDER TO BE IN CLASS LIST!
Keep at it **CONTINUOUSLY** if there is any problem. Have to show ID!!! Show MAP, find **Apply** in Guide, And so on . . .

BRING UNIX GUIDE/BOOK (Glass), ESPECIALLY K&R, TO CLASS FROM NOW ON.

GET BUDDY PHONE # NOW, IF MISS CLASS -- CATCH UP.

This class starts slow because entering student experience is highly variable. If you find it too easy, wait a few weeks. TAKE ROLL.

HW 1. Read K&R 1.1 -> 1.9, Ch 1 (20 min), Ch 2 (details) of Glass UNIX, All of UNIX Guide. Find hw0 in UNIX at ~poneil/cs240/hw0/assignment.(explain directory tree structure): This is NOT Web accessible!!

Create hw0 directory of your own under existing cs240 directory, place assignment and solution files there.

One task is to create hello.c file in hw0 directory, use vi editor (covered in UNIX Guide). File hello.c is first C program in Section 1.1 of K&R. (Next pg.):

```
#include <stdio.h>

main( )
{
  printf("Hello, world!\n"); (\n is special char - see P 193, K&R for others)
}
```
(Allow class time to find it in K&R. Slight difference.)

After create with vi editor, compile and run with following commands:

```
gcc  hello.c      (Compiles program, creates a.out run program file)
a.out             (Runs it)
```

Turn in assignment as **script**. Following command sequence for turn in:

```
script      (starts script of all input & output to typescript file)
ls -l       (lists files in directory, with timestamp last update)
cat hello.c       (displays the source file)
gcc hello.c
a.out
exit        (ends typescript)
```

Print out typescript and hand in hardcopy in class. Leave THAT copy of typescript online (proof finished on time)

How to use vi editor, as in Guide (Look up in Guide - demonstrate index)

```
vi hello.c  (first time, creates file)
```

3 modes: command, insert, and last-line command modes
Idea of mode: program expects certain kind of input, like UNIX command
Enter vi always in command mode -- assume text exists -- commands are:

```
x  to kill char under cursor;    dd to kill line cursor sits on,
6x (for example) kills 6 chars, 3dd kills next 3 lines,
arrows:  ->, <-, up-arrow, down-arrow /  if missing on home terminal,
  CNTRL-chars exist which are equivalent -- see Guide
```

There are a number of commands to enter insert mode

i - insert before cursor (Now is the time for all good . . .; type i, x, get:
Now is xthe time for all good . . .)
a - insert after cursor (a, z, gives: Now is tzhe time for all good . . .)
Need i to insert new char before line, need a to insert new char after end.
O - create new line before current line, start insert
o - create new line after current line, start insert

To leave insert mode, return to command mode type ESC (example of a mode that you have trouble leaving if you don't know the key).

last-line commands for commands where character string must be typed, type colon (:) and see (:) on last line of screen, ready for input.

:w filename	-	write contents of vi buffer to file "filename"
:w	-	don't need to name file if writing to file you're editing.
:r filename	-	read file "filename" into current buffer (at end)
:q (or :q!)	-	quit back to UNIX without saving updates to file

The command ZZ (CAP-Z CAP-Z) in command mode will quit back to UNIX and save updates to file.

Too many commands in vi (Type your name and see what happens!), but note a few special commands:

/string	search forward to find "string"
5yy	cut out (copy) next 5 lines (any digit)
p	paste (what just copied) after current cursor line

Go over basic UNIX commands in Guide. Responsible for them next time.

Class 2.

New handout needs? HANDIN. GET BUDDY in class, phone # to find out details of missed class!! **YOU MUST APPLY**, even if you already have a UNIX Account! Keep at it now CONTINUOUSLY -- your responsibility.

Assignment hw1 now online or soon will be if you want to get head start. Glass UNIX, Ch 1 (20 min), Ch 2 (know all commands), K&R1.1 -> 1.9

DON'T EVER miss class to finish homework. Time of last update for script output is given by ls -l command, grader can check, and midnight of due date is ON TIME. After that time, 1 point off out of 10 if get in by NEXT CLASS. After that I put answer on-line, so only programs finished before that time can count (maybe some are finished). I drop bottom 2 hw grades.
Have all your books? Any Questions? **ROLL. Money for Notes: $5.00**

Example program from pg 15 of K&R -- put on board

```
#include  <stdio.h>     /* copies definitions from library file for I/O funcs */

#define LOWER 0        /* Symbolic constant                                   */
#define UPPER 300      /* Idea is, in large program don't want to write       */
#define STEP 20        /* 20, hard to edit (other meanings for 20)            */

  /*  Print out Fahrenheit - Celsius conversion table, in steps of 20 C        */
main( )
{
   int fahr;                    /* declarations - MUST come first in C        */

   for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
      printf ("%3d, %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

(GO SLOW NOW!)
for loop parts: for (initialize; loop test; increment) -- Acts like Java:

```
   fahr = LOWER;          -- initialize
   if( ! (fahr <= UPPER)) -- if loop test fails then (see the !)
      goto next;          -- end loop
body:  printf( . . . );    -- body of loop
   fahr = fahr + STEP;    -- incremental step
   if (fahr <= UPPER)     -- loop test -- if loop test succeeds
      goto body;          -- go execute body of loop again
next:  . . .              -- otherwise fall through loop                */
   for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
            ^initialize^loop test          ^increment
```

But very general -- e.g., incremental step can MULTIPLY instead of adding, loop test can test for entirely different condition than <= (SAME AS JAVA)

Now statement not in JAVA: printf ("%3d, %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
Consider instead: printf ("Values:Δ%3d,Δ%6.1f\n", fahr, (5.0/9.0)*(fahr-32));

Where the characters specified by "Δ" are what we write to show a space explicitly. Quoted string "Value: . . ." is just like "Hello, world", but %3d and %6.1f are special placeholders, called conversion specifications.

This means that the two expressions following the quoted string, fahr, and (5.0/9.0)*(fahr-32), are to be printed according to the prescription given. The table here would look like:

```
Values:ΔΔΔ0,ΔΔ-17.8
Values:ΔΔ20,ΔΔΔ-6.7
Values:ΔΔ40,ΔΔΔΔ4.4
 . . .
```

Other characters in "Values: . . .", such as "," and "Δ" are printed literally.

The %3d means that an integer is printed so as to take up 3 spaces, right adjusted -- "Δ40", but no initial space for "100" -- still have space before 100 because came after "Values:Δ".

The %6.1f means to print a float number (floating point or “double” by default, represented with a fractional part), with a total of 6 spaces used up and 1 digit after the decimal point: thus "Δ-17.8" uses 6 spaces.

See P 154 in K&R for a table of "conversion characters" like %3d and %6.1f. (Allow class time to find it.) These pages in K&R are important, need in open book Exams/Quizzes. So far have mentioned P 193 (\n) and P 154.

Note (5.0/9.0)*(fahr-32) prints with a fractional part: although fahr is int, and (fahr-32) is int, (5.0/9.0) is double (floating point), and when we multiply a double by an int we get a double to preserve accuracy (see Section 2.3).

Note that formula is right: if fahr = 32, centigrade should be 0, if fahr = 212, centigrade should be 100.
New point. Idea of standard input (stdin) from keyboard and standard output (stdout) to terminal screen. (**NO stdin stdout IN JAVA**. Hard to control I/O) Will now write a program copy.c to copy stdin to stdout. If compile it:

gcc copy.c -o copy (executable file called copy now, like a.out before)

Can run copy, typing "copy" on a new line, and will echo all characters you type on the screen. Doesn't seem like much, but this is the basis of what a command interpreter does.

By the way, the only way you can get it to terminate is by typing a CTRL-D (ON A NEW LINE !) which is interpreted as an End Of File. (Stole a character, means you can't echo a CTRL-D after a New Line).

We can "redirect" stdin and stdout. Take stdin characters from a file named "copy.in" instead of keyboard:

copy <copy.in

will type out chars from copy.in to screen (acts like 'cat command'). Can also put stdout characters to a file named "copy.out" instead of terminal screen:

copy >copy.out

When you type this, the command interface (UNIX Shell) will create the file copy.out, overwriting any previous version, and redirect stdout to that file before the copy program ever gets control. This command will take all characters you type and put them in a file, copy.out, kind of like a cheap editor. Finally, you can copy one file to another by giving command:

copy <file.in >file.out (acts like 'cp command') Now let's see the program, copy.c.

```
#include <stdio.h>
main( )                         /* copy stdin to stdout */
{
  int c;                        /* c will hold characters */

  c = getchar( );
  while (c != EOF)  {
     putchar(c);
     c = getchar( );            (LEAVE THIS ON BOARD WHILE TALK)
  }
}
```

getchar( ) and putchar( ) are functions which are defined (as macros) in stdio.h. The function getchar( ) (with no argument) gets a character from stdin, returning the int value of the ASCII code; putchar(c) puts the int-eger value c (argument) out to stdout as a character (nothing is returned).

Go over logic -- what happens.

Page 12 in these notes: values 0-127 decimal are ascii codes characters. Fits in one byte: 01111111 (binary) is 127. (1 = 2-1. 1+2 = 4-1. 1+2+4 = 8-1 . .).

int type c uses four bytes of significance, from -2^31 to 2^31 -1. Have another integer type, called **char**, which holds only 1 byte of significance from -128 to 127 (or else 0 to 255), unlike Java (16 bit Unicode). Why didn't we use type char c in this program instead of int type? (Wait for answer.)

Because of EOF which must be a special integer value, different from any normal character (some char sets go from 0 to 255). Needed more significance.

Now this form is not exactly the way a C programmer would want to write it. (rewrite on side of pgm:)

```
c = getchar( );                          while (( c = getchar( )) != EOF)
while (c != EOF)  {        --->                putchar(c);
putchar(c);                          (Expression above OK in JAVA but no getchar().)
c = getchar( );
}
```

Secret with new form is that an assignment statement (c = getchar ( )) in C is thought of as an expression, which itself has a value, equal to the value assigned to the left-hand target variable.

Now evaluate the While from the most nested expression out. Bring in each stdin character in testing TRUE, put it out in body of loop. Value of this economy is that compiler probably compiles more efficiently.

Why not: while (c = getchar( ) != EOF) ? What does 6 + 2 * 5 mean? How about (6 + 2) * 5? Go over table of precedence P 53 of K&R. What would happen if rewrote while ( ) as above?

Class 3.

Turn in hw0. Hw1 assignment file on line in ~poneil/cs240/hw1.
DON'T USE <string.h> — WRITE YOUR OWN, e.g. strlen(s), pg 39 mentions

Anybody not yet have UNIX ID? Don't leave until talk with Operators, check twice a day get ID, do first assignment. (Won't count as late.) Catch up.
Everyone have all books? With them? CLASS NOTES! Any Questions? Start reading Chapter 2 of K&R. **ROLL. Money for Notes**.

What does this do?

```
main( )
{
   int c, m;

   m = 0;
   while ((c = getchar( )) != EOF)      /* Counts lines in a file ASSUMING */
   if (c == '\n')                       /* each line ends with \n — if ends */
      ++m;                              /* with EOF, not counted */
   printf ("%d\n", m);
}
```

Note: if (c == '\n'); Use ==, >=, <=, !=, only single comparators are >, <. In particular, c = '\n' is NOT A COMPARISON. It is an ASSIGNMENT statement.
If we use it, c will be set equal to '\n', and value will be TRUE, since int value is not zero. No Boolean types in C: integer zero is FALSE, integer non-zero is TRUE. c == '\n' is set to zero or one, according to truth.

++m is an expression which when evaluated has the effect: m = m + 1; Recall that an assignment statement gives a value; here is a seeming expression which has the effect of an assignment statement.

Statement forms we have had so far:

```
while (logical expression)
   statement;

for (initialize;  loop test; increment)
   statement;

if (logical expression)
   statement1;
[else                                   /* optional else */
   statement 2; ]
```

Arrays

Idea of character string (Section 1.9). C language (unlike Java) has no "special" string type. Character string,must have an array of char type values (integer values) ending with '\0' = NULL (NULL character) = integer value zero.

Shorthand for initialization, printf, use quoted strings:
char msg[ ] = "hello\n"; (NOTE DOESN'T WORK AS ASSIGNMENT STMNT!)

h	e	l	l	o	\n	\0

```
 #include <stdio.h>

/* count digit chars 0-9 coming from stdin */  (simpler than on p 22 K&R)
main ()
{
   int c, i;            /* c for getchar - ASCII code for characters              */
   int ndigit[10];          /* Note:  subscripts 0 . . 9 -- 1 for each digit      */

   for (i = 0; i <= 9; ++i)
      ndigit[i] = 0;         /* initialize to zero                                */
   while ((c = getchar()) != EOF)
     if (c >= '0' && c <= '9')    /* condition: if c is a digit — && is "and"    */
      ++ndigit [c - '0'];    /* SLOW:  '0' is an integer                         */
   printf ("digits = ");
   for (i = 0; i < 10; ++i) /* print out counts of 10 digits                     */
     printf ("%d  ", ndigit[i]);   /* Note no \n, so all on 1 line               */
   printf ("\n");
}
```

Go over program with several functions on pg 29 (see next page). First job is to take a series of lines from stdin, save the longest line, and print it out to stdout when stdin comes to EOF. (Copy onto board.

=> Go over logic. Point out function calls in main. Note len returned by getline won't be zero until end of file, because counts 1 for '\n' character, even if that is the only one, with two newlines in a row.)

Bunch of new ideas, including idea of functions, "%s", order of evaluation in X && Y && Z. Note will use getline in homework: trim, 1.18, 1.19.

printf ("%s", msg) keeps copying letters in msg to stdout until runs into '\0'. More on this next time. Show how to program Visible Typewriter.

Program from pg. 29 of K&R

```
#include <stdio.h>
#define MAXLINE 1000

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* print longest input line                                                    */
int main()
{
    int len, max;                                   /* current/max line length  */
    char line[MAXLINE], longest[MAXLINE];           /* current/max line         */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)  /* only fails test at EOF       */
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0)                                    /* there was at least one line */
        printf("%s", longest);
    return 0;   /* should have int main() above    */
}

/* getline: read a line into s[], return length    */
int getline(char s[], int lim)
{
    int c, i;

    for (i = 0; i < lim-1 && (c = getchar())!= EOF && c != '\n'; ++i)
        s[i] = c;                                   \ Note order of evaluation above!
    if (c == '\n')  {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy: copy string in array from[] to array to[]; assume to[] is big enough. */
void copy(char to[]; char from[])
{
    int i

    i = 0;
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Class 4.

HW: Reading Chapter 2. Hw 1 on-line. Quiz on Class 8 (MAY BE OVERRULED IN CLASS ANNOUNCEMENT). Soon cover binary numbers. Questions?

In trim.c, trailing blanks means comes before newline: trim <trim.in .trim.out
Use od -x to see non-printing chars in trim.in and trim.out.
od -x trim.in Form of output is:

```
00000000  0909 4e68  . . .  2020
00000020 2020 200a
```

Reason we use hex. Say string ends with four spaces, in hex 20202020. Translate this to octal!!?? Characters keep changing representation!!

In writing visible typewriter, job is to read in character c, using getchar

c = getchar() Put this in loop as we've seen in class and book

Print out ASCII name and hex value (ASCII value is simply int value held in c, use conversion specification "%x" for hex): (WHERE FIND '%x' IN TEXT? pg 244.)

```
printf ("%x  ", c);  /* NOT the same as putchar(c) -- prints the letter! */
```

But we need to look up NAME of ASCII character somewhere. See page 11 for these names. We will create an array of char type with ALL these names.

```
char asciiname[ ] = "NUL\0SOH\0STX\0ETX\0"
"EOT\0ENQ\0ACK\0 . . .                    "
 .  .  .
"   .   .   .   .   .  \0ΔΔ|\0ΔΔ}\0ΔΔ~\0DEL\0";
```

Note that this form initializes the asciiname array, so that the code for 'N' goes in asciiname[0], 'U' in asciiname[1], 'L' in asciiname[2], the zero value represented by '\0' in asciiname[3]. We have now given the first asciiname. Starting the second, we have 'S' in asciiname[4], and so on.

The initialization form: char s[ ] = "Hello," " world"; will concatenate the two strings "Hello," and " world", so here is a convenient way to continue a single string when we run out of room on a line. The initialization line is not complete until a semi-colon appears at the end of the last line.

Here is the **ASCII Encoding**, a correspondence of keyboard characters with integers from 0 to 127 (0x7F in hexadecimal, 0177 in octal)

char	hex	oct
NUL	00	000
SOH	01	001
STX	02	002
ETX	03	003
EOT	04	004
ENQ	05	005
ACK	06	006
BEL	07	007
BS	08	010
HT	09	011
NL/LF	0A	012
VT	0B	013
NP/FF	0C	014
CR	0D	015
SO	0E	016
SI	0F	017
DLE	10	020
DC1	11	021
DC2	12	022
DC3	13	023
DC4	14	024
NAK	15	025
SYN	16	026
ETB	17	027
CAN	18	030
EM	19	031
SUB	1A	032
ESC	1B	033
FS	1C	034
GS	1D	035
RS	1E	036
VS	1F	037

char	hex	oct
SP	20	040
!	21	041
"	22	042
#	23	043
$	24	044
%	25	045
&	26	046
'	27	047
(	28	050
)	29	051
*	2A	052
+	2B	053
,	2C	054
-	2D	055
.	2E	056
/	2F	057
0	30	060
1	31	061
2	32	062
3	33	063
4	34	064
5	35	065
6	36	066
7	37	067
8	38	070
9	39	071
:	3A	072
;	3B	073
<	3C	074
=	3D	075
>	3E	076
?	3F	077

char	hex	oct
@	40	100
A	41	101
B	42	102
C	43	103
D	44	104
E	45	105
F	46	106
G	47	107
H	48	110
I	49	111
J	4A	112
K	4B	113
L	4C	114
M	4D	115
N	4E	116
O	4F	117
P	50	120
Q	51	121
R	52	122
S	53	123
T	54	124
U	55	125
V	56	126
W	57	127
X	58	130
Y	59	131
Z	5A	132
[	5B	133
\	5C	134
]	5D	135
^	5E	136
_	5F	137

char	hex	oct
`	60	140
a	61	141
b	62	142
c	63	143
d	64	144
e	65	145
f	66	146
g	67	147
h	68	150
i	69	151
j	6A	152
k	6B	153
l	6C	154
m	6D	155
n	6E	156
o	6F	157
p	70	160
q	71	161
r	72	162
s	73	163
t	74	164
u	75	165
v	76	166
w	77	167
x	78	170
y	79	171
z	7A	172
{	7B	173
|	7C	174
}	7D	175
~	7E	176
DEL	7F	177

Note that single asciiname names, such as }, have two spaces before them so that all names take up 3 characters followed by a \0. (Put spaces before, or strange things can happen.) A \0, although two marks as we see it, will be interpreted by the compiler to stand for a single character, a NULL (value zero), placed in the array at that position.

Printf expects to receive a pointer to a (format) string of char values, types out characters with these ASCII codes until it reaches a NUL. The full printf statement to print out the hex ASCII code and the ASCII name is:

```
                                / '&' is magic - creates pointer to array position.
  printf ("%x  %s", c, &asciiname[4*c]);
```

We subscript asciiname by 4*c, because there are 4 chars to each ASCII name, e.g. Try c = 0. See print out NUL. Try c = 4, show prints out EOT.

There are a few characters which can't simply be placed in the quoted initialization string as they stand, for example " and \. How do we handle this?

" . . . \0ΔSP\0ΔΔ!\0ΔΔ"\0ΔΔ#\0 . . . " (" is hex 22)

Problem is that compiler will interpret " character as the end of string. We need to indicate that it is quoted, that it is to be taken literally by the compiler as an ASCII value, and the way to do that is precede it by a \.

" . . . \0ΔSP\0ΔΔ!\0ΔΔ\"\0ΔΔ#\0 . . . " (" is hex 22)

Now even though there seem to be five chars in that block, ΔΔ\"\0, it is not true because \" is a single character value, just as \0 is.

What other special characters like this need this special treatment? See pg. 193!

How to debug. See http://www.cs.umb.edu/helproot/cs240/cs240.html and UNIX Guide. Talk about environment and what want to do. A debugger is what a professional programmer uses, rather than putting in lots of printf statements to track down a bug. Start with right compiler option.

```
gcc -g  vt.c  -o  vt  (-g for debug,  -o replaces a.out with filename vt)
```

See UNIX Guide. Now, instead of just typing name of program, use

```
gdb  vt
```

Gives message; Ready to run -- not yet running. Want to interact with running pgm, not just let it run free to end. Type:

```
b main      (break at main(); will stop running when it comes to main() in
    execution -- often lot of things done first)
r  <vt.in    (run, taking stdin from vt.in -- less confusing)
```

Will stop when encounters main(). Now can single step through program, s or n (skip entering called functions), put out values of variables. Examples follow.

```
p  i   (print value of variable i)
p  3*i   (print value of expression 3*i)
p/x  i   (print in hex format value of variable i)

i  lo  ("info" - give values of all local variables)

h      (help -- pretty good messages -- lists topics)
h  topic    (help on named topic)
h  p  (help on command p for printf)

q      TO QUIT (leave debugger)
```

More complex stuff in UNIX Guide. Setting breaks at line numbers: b 36, b fn.c:22 if i == 3. Getting line numbers from "list" or "l" command: l 22, print 10 lines around line 22 in main, l fn.c list first five lines, then l means next 10 lines. i b, to get info on breakpoints, d 3, to delete bkpt 3. c for continue after bkpt encountered. **Good QUIZ Questions.**

Starting Chapter 2 next time.

Class 5. HAND OUT PRACTICE QUIZ.

QUIZ soon (Class 8). 3 short questions, time limit 20 minutes, open book. Guarantee question on UNIX, might be something only MENTIONED in UMB UNIX Guide (I'm not covering UNIX, but you're responsible).

Turn in hw1 soon. Full credit until Midnight, then 1 point off out of 10 until next class midnight. After that answer is put on-line. Can't get credit.
Starting Chapter 2 K&R this time. Read ahead.

Hw 2 is on line soon/now. 2 weeks, but rather a lot to do. Questions?

Call by reference vs. call by value.

Simple variables passed as function parameters in C are actually only **copies on the stack.** (Give short idea of stack, function call).

This is known as Call by Value. Unlike some languages, which have call by reference, can't make changes to arguments in call by value -- just change a copy on the stack (Explain). To allow changes, must pass pointers to variables. Give an idea here, but you're not responsible for pointers until later.

The following function

**** DOESN'T WORK ****

```
int exchgint (int a, int b)
{
  int dummy;

  dummy = a;
  a = b;
  b = dummy;
  return 0;
}
```

(**NOTE**: No pointers in JAVA)
**** POINTER VERSION INSTEAD ****
(Read "int *pa" as: the thing pointed to by pa is an int -- pa is a pointer!)

```
int exchgint (int *pa, int *pb)
{
    int dummy;

    dummy = *pa;
    *pa = *pb;
    *pb = dummy;
    return 0;
}
```

Note, array names automatically passed by pointer; you don't have to create a pointer yourself. (**Recall**: copy(longest, line)

Idea of Scope and Duration of a variable. Look at K&R pg 32, slightly different definitions of variables from what we saw in 1.9.

Note that variables named i in both getline and copy, but they are different: they are private to the routines where they are declared.

Each local variable of a function (inside { }) comes into existence only with its call, disappears when a return is performed. (The same can be said of the arguments of a function – copied in call by value.) TALK ABOUT STACK! Local variables are said to be automatic.

There is an alternative to this, known as static. A variable declared in a function can be in durable memory if it is specially declared. E.g., the seed of a random number generator so it will have memory from one invocation to the next and not always give the same random number.

```
int rand( )
{
  static int seed = 1;     /* initialize to 1, remember between calls */
  . . .
```

The alternative to local is external.

The declarations you see on page 32 of K&R, for char line[ ] and char longest[ ], are **external** (sometimes called global): they are outside any function, unlike on page 29 when they were inside main( ) { . . . }, the main function.

When variables are declared outside any function like this, they are visible to all functions in the file (there may be more than one file, and they must be specially mentioned to be visible from a different file).

They are also automatically stored long-term, rather than automatic.

Advantage in this definition of getline, copy is don't have to pass variables "line" and "longest" as parameters -- simpler calls E.g., function "copy" no longer has "from" and "to" parameters.

Disadvantage is loss of flexibility -- copy no longer works for any 2 arrays. Normally more important than simplification.

We will talk more about all this in Chapter 4. We have now finished Chapter 1. Now starting Chapter 2.

Data types: char, int, float, double. A double variable is a float with more significance -- both represent numbers with fractional parts. Don't do much with float in this course.

Much of what follows is different in C and Java.

In C, A char variable is an int with less significance. char has 8 bits (one byte): $-2^7 \le$ char $\le 2^7-1$. (**in Java, char is 16 bits, Unicode, NOT an int. Java byte is 8 bit int.**))

In C, with signed numbers (**Java has ONLY signed numbers!**), the leftmost (sign) bit is on for negative numbers and off for positive numbers. (This does NOT mean that you turn the leftmost sign bit on the get the negative of a positive number.) Thus the largest positive char value in C is 01111111 = $2^7-1 = 127$. See smallest negative shortly.

The type int has 32 bits on our machine, $-2^{31} \le$ int $\le 2^{31}-1$. There are also two other possible types: short int and long int. a short int on our machine has 16 bits, $-2^{15} \le$ short int $\le 2^{15}-1$, and a long int, again on our machine, is the same as an int. (**In Java a long int is 64 bits on all machines.**)

On general machines, we are only guaranteed in C that char has 8 bits and length(short int) ≤ length(int) ≤ length(long int).

In C, there is also another kind of orthogonal type declaration, unsigned. (**There is no unsighed type in JAVA.**) An unsigned int is interpreted as having no negative numbers; therefore we have: $0 <$ unsigned int $< 2^{32}-1$ (instead of: $-2^{31} <$ int $< 2^{31}-1$).

A char type may or may not have negative values (it is implementation dependent) but an unsigned char certainly doesn't.

The hardest thing to realize is that constants have types in C (**and Java**). This is important in evaluating expressions. (3./2.) is equal to 1.5, but (3/2) is equal to 1 and has no fractional part. (Recall fahr to cent pgm, p15, (5.0/9.0)) Now what value would you say the expression (3/2.) has?

Here are C constants and their types:

1234	int
1234L	long int
1234UL	unsigned long int
1234U	unsigned int
123.	double (decimal point)
1e-2	double (because of e, equal to .01)
037	37 octal (leading 0 -- base 8) = 3*8+7 or 31 decimal (int)
NOTE: First character is a ZERO, not the letter "Oh". 037 == 31	
0x37	37 hexadecimal (base 16) = 3*16+7 or 55 decimal (int)

'a' char, integer value (range 0-127) of ascii code for the letter a (see these Notes pg. 11)
'0' char, integer value for digit 0
'\b' char, integer value of backspace char (K&R Pg 193)
'\123' char with value 123 octal (or 55 hex: '\x55'), == 'U'
'\ooo' char, any three octal digits with value in range 0-255 decimal (0377)
'\xhh' char, any two hex digits with value in range 0-255 (0xff)

Note that char constants are nothing special except that they are stored as 1 char integers and follow certain rules of conversion (in any expression, such as: if (c=='0')); octal and hex don't even cause the constants to be stored differently -- we could just as easily use decimal.

Conversion of char type constant is usually the same as if we had used int. However, we use char constants to remind the user that the constant is for a particular character or is in the range 0-255. A hex number is often used because it is important to picture a certain bit pattern.

Different bases for numbers: Binary, Octal, Hexadecimal

OCTAL	BINARY	OCTAL	BINARY
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

HEX	BINARY	HEX	BINARY	HEX	BINARY	HEX	BINARY
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

0x7A (or '\x7A') = (in binary) 0111 1010 (digit 7 followed by digit A)
Each hexadecimal number corresponds to a group of 4 binary digits
A bit pattern can be specified in hexadecimal easily
0x39 = '\x39' = (in binary) 0011 1001

octal numbers are groups of three octal digits (START FROM RIGHT)

0132 = '\132' = 001 011 010 (**NOTE!** Underlined part is 8-bit char)

I can reblock this in groups of four (starting from the right) to get:

0132 = 001 011 010 = 0 0101 1010 = 0x5A

Check this: 0132 = 1*8^2+ 3*8 + 2 = 90 (base 10)
0x5A = 5*16 + 10 = 90.

Signed numbers. To get a negative form of a binary number, flip all bits and add 1 (in binary). Say start with value 0x55 for signed char variable X and want to write -X. I will work with 8 bit char integers. 32 bit (int) form is 0x55, 8 bit (char) form is '\x55'

```
'\x55'   =            01010101                (8 bit char number)
flip bits: ~'\x55' =  10101010                (ONES COMPLEMENT)
   add 1              10101011  =  '\xAB'     (TWOS COMPLEMENT)
```

We say '\xAB' = -('\x55'), to mean that when we add these 2 numbers in binary:

```
   01010101
+  10101011
 -----------
(1) 0000000    We get 0 in the rightmost 8 bits (where the number is)
  \- Carry Bit
```

Definition of -X is that X + -X = 0 in 8 bits (with a carry out on the left side, which is however not one of the 8 bits.) So -X is the same as
(~X + 1). Try testing this in a program!!! Note when we just flip all bits, X to ~X, called 1's complement, when take (~X + 1), called 2's complement.

```
Note:  -(-X) = X.  True?  X =    01010101  (FLIP:  10101010, add 1)
                          -X =   10101011  (FLIP:  01010100, add 1)
                        -(-X)=   01010101
```

Now note: if X = 01111111, FLIP: 10000000, add 1)
-X = 10000001
But then since X is 127, -X is -127, have 10000000 is even more negative (-X - 1). 10000000 is **most negative number**, -128.

if X is 10000000, what is -X? FLIP: 01111111, add 1, 10000000.
Problem on this in hw.

Note we could have predicted that such a number exists. Even or odd number of 8 bit numbers? (Even.) But does every integer have a different negative? (No. not zero.) But then another number must have a property like this, since even number of integers total.

Class 6.

Hand out Practice Quiz.
Turn in hw1 (by midnight). Hw2 on-line. **Quiz 1 will be class 8.**

Recall 2's complement definition of -X from last time. IN PARTICULAR NOTE that although there is a sign bit:

0 1000101 (positive number (sign bit 0) = '\x45' = 69_{10})

One DOES NOT get the negative of this by turning sign bit on ! In general, must flip bits and add 1. (Do this, value is -69.) Show that X + ~X = 11111111. X+~X+1 = 00000000. Therefore ~X+1 is equal to -X)

If we did turn on sign bit in above, 11000101, what would we get? It's a negative number, not -69. Try flipping bits, 00111010, add 1, 00111011, result is '\x3B', value is 48+11 = 59_{10}. Thus 11000101 was -59 !

Note that if now do this again with 00111011, flip 11000100, add 1, get 11000101, the original number, back. Always get same number back: -(-x) = x. Except if x = 0, get it back right away. 00000000 -> 11111111 add 1, get 00000000. (Sure! -0 == 0)

Now there are an even number of binary numbers in k bits. If x -> -x -> x is a closed loop for all pairs of numbers except 0, seems to be one left over. Indeed there is. What are successively smaller negative numbers?

number X	-X	
00000000	00000000	Note successive values of X
00000001	11111111	achieved by adding 1, successive
00000010	11111110	values of -X by substracting 1.
00000011	11111101	With -X values, if replace 1's by
00000100	11111100	0's, starts with all 0's one line
. . .	. . .	later and adds 1's.
01111110	10000010	
01111111	10000001	
	10000000	

There seems to be no positive for smallest negative; char values go from -128 to +127; int from -2^31 to 2^31-1. What happens if take negative of 10000000? Get back 10000000 (like 00000000). Homework Problem on this.

Note that humans think in terms of decimal, or after a bit of practice in terms of binary or hex or octal. The computer thinks in an extremely large base -- sees a pattern of bits (even 32 bits) all at once, as if it knew 2^32 digits. It is willing to convert to help feeble minded users in printing out: %d, %x, %o (no binary format).

/-Don't Need subscript 10

How to convert decimal to binary. From 117_{10} get binary by halving the number successively (rounding down) and writing down the digit 1 when have an odd result, 0 when result is even (consider number itself the first result):

117	1	LSB
58	0	
29	1	
14	0	
7	1	
3	1	
1	1	MSB

Then read UP the bit sequence: 01110101 = '\x75', or in base 10 have 7*16+5 = 112 + 5 = 117. Do it with another: 203. Works because dec # ends with 1 of odd; now divide by 2, to work next digit left.

	bin		bin
from any one of:	oct	to any one of :	oct
	hex		hex
	dec		dec

From bin to oct or hex, block in groups of 3 or 4 from the right. From bin to dec, sum up powers of 2 (01110101 = 1+4+16+32+64, or else go to hex first and sum up multiple powers of 16. Thus, hex or oct to dec.

From hex or oct to bin, write down digits in blocks of binary digits from the right. From hex to oct or reverse, go through bin.

From dec to bin, use algorithm. From dec to hex or oct, go through bin.

Bitwise operations. PROMISE LOTS OF EXAM QUESTIONS ! ! !

Ops: ~ (unary not), & (and), | (or), ^ (xor), << (leftshift), >> (rightshift)

unsigned char n = '\xa6'; binary 10100110 (Why unsigned? Will see.)

~n :01011001 (just the 1s complement: flip bits, 0 to 1, 1 to 0.)

```
n | '\x65':   10100110
          |   01100101   turn on bit in result if it was on in either operand
              11100111


n & '\x65':   10100110
          |   01100101   turn on bit in result if it was on in both operands
              00100100

n ^ '\x65':   10100110
          |   01100101  turn on bit in result if was on in exactly 1 operand
              11000011
```

Call ^ the exclusive or (xor) operator - what is sometimes meant when we say "or" -- one or other but not both.

```
  '\x18' :          00011000
  '\x18' << 1       00110000  shift 1 to left (like multiply by 2)
  '\x18' << 2       01100000  shift 2 to left (like multiply by 4)
  '\x18' << 4       10000000  shift 4 to left (bits disappear off left)
  '\x18' >> 2       00000110  shift 2 to right (like divide by 4)
  '\x18' >> 4       00000001  shift 4 to right (bits disappear off right)
```

If start with '\xa5', 10100101, and shift right result depends on whether containing variable is signed or unsigned char:

```
  '\xa5' :          10100101
  '\xa5' >>2 :      00101001  (bring in 0 on left, like divide by 4 of UNSIGNED)
  '\xa5' >>2 :      11101001  (bring in 1 on left, like divide by 4 of SIGNED)
```

Bringing in 1 on the left is called "sign extension". Negative number divided by 2 or 4 is still negative number' '\xa5' >>2 = 11101001 = '\xe9'. (But this is not true on all machines: See K&R pg. 49, lines 8-10.)

We can work this out to check that the proper division is occurring. '\xa5' is minus what? 10100101 and '\xe9' is minus what? 11101001

Flip bits	01011010	Flip bits	00010110
add 1	01011011	add 1	00010111

So it's minus 0x5B, or minus 5*16 + 11 = -91.

So it's minus 0x17, or minus 16+7 = -23.

Note -91/4 = -22, -91%4 = -3 (or -91 = 4*(-91/4) + -3). When divide negative number, remainder must be negative, not positive (ALWAYS rounds DOWN!)

Means (-1)/2 = -1. Try it in a program loop! Not the same as -(1/2).

But note that if the number being divided is UNSIGNED (e.g., an unsigned char with value 'x\a5'), then >> will NOT bring in 1 bits on the left!

Note that there's an operator in Java, which has no unsigned type: the operator >>> acts like >> in all cases except that it doesn't sign-extend for negative numbers: brings in 0's on the left, even if sign bit is 1.

Now a new question: given char n, how to turn off all bits except the bottom 5: n = n & '\x1f'. E.g., if n = '\xa5' or 10100101 binary

```
n & '\x1f':    10100101
          &  00011111  turn off all bits except bottom 5
             00000101
```

called "masking" the bits -- only see bits on where 1's found in mask constant.

Now, how to turn on the highest two bits in char n (if already on, leave it on). n = n | '\xc0'. '\xc0' = 11000000. Assume n = '\xa5'

```
n & '\x1f':    10100101
           |  11000000  turn on top two bits
              11100101
```

Idea is this. Use bits in char n as flags. Turn on the flags in n corresponding to bits on in '\x48. Test if any of the flag bits '\x64 are on.

Show that x ^ (x ^ y) == y. Do a real example!

Note that x = x & ~077 (octal for a change), SETS LAST 6 BITS in x to 0. If don't know size of x (number of bytes in int), ~077 will become: ~00 . . . 00111111 or 11 . . . 11000000, of whatever size is needed (extends itself with 1 bits on left for length of x).

Class 7.

QUIZ 1 Next Class.

Other constants. String constant: "I am a string." Is an array (a pointer to a string) of char values somewhere (where is the memory location for the constant 6?), ending with NUL = '\0', the char with zero value. Examples:

printf("I am a string.\n") or declare: char msg[ ] = "I am a string";

Naturally, "0" is quite unlike '0'. The value "0" can't be used in an expression, only in things like printf(). (Can't write "This " + "that"; just write strings next to each other to catenate: "this" "that" same as "thisthat".)

Have string functions in C that work with these nul terminated char arrays:

#include <string.h> (See pg. 241, Appendix B, then B3, pg. 249)

With this can use: len = strlen(msg); where msg is string in a string array.

IN ALL HOMEWORKS, EXAMS, WRITE YOUR OWN STRING FUNCTIONS. (This is just until you know how they work. At end of this course, you will be expected to know and use existing functions, not write your own.)

Logic of strlen code is like:

```
int strlen(char s[ ])  {
   int i = 0;

   while (s[i])
      i++;
   return i;
}
```

Enumeration constants mentioned in K&R.

enum boolean {NO, YES}; (starts from 0 and increment)

Now can declare a variable of type boolean: (**No Java built-in Boolean.**)

enum boolean x;

Later can write x = NO;

Just a shorthand for creating symbolic constants with #define. Except that if define months as type with values 0, 1-12, (0 = ERR, 1-12 = JAN-DEC) declare x of type months, debugger might type out value FEB for 2.

Idea of "const" declaration (like "final" in Java): warns compiler string shouldn't change. Commonly used for function parameter. For example:

```
const char msg[ ] = "Warning: . . .";

int copy(char to[ ], const char from[ ]);
```

Now if logic of copy function attempts to modify the from string, it will fail (the exact form of failure is installation defined).

Operators. Arithmetic: +, -, *, /, %

```
int x, y;

x / y truncates (no fractional part)          EXAMPLES
x % y is the remainder when x is divided by y.
Always true that:   x == y*(x/y) + x%y

int year;                                              "or"
                                                       /
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)  (See pg. 41.)
   printf( "%d is a leap year\n", year);
else
   printf( "%d is not a leap year\n", year);
```

See precedence table, P 53, for why inner parens above NOT necessary.

We call a comparison between two arithmetic expressions a "relation"

```
ae1 <= ae2  (comparisons:  <, <=, ==, !=, >=, >  )
```

A relation is either true or false (1 or 0) for any given ae's.

```
if ( i < lim-1 == j < k)   (What's it mean?  See precedence P 53)
```

Logical expressions are built up out of relations using &&, ||, ! (not):

```
Instead of c != EOF, could write ! (c == EOF)
```

Type conversion. Consider expression involving different types:

```
/* atoi: convert character string of digits to int (base 10) */
   (e.g., atoi stands for ascii to int.)
int atoi(char s[ ])              (Also have itoa, atox, atob)
{
   int i, n;

   n = 0;
   for (i=0; s[i] >= '0' && s[i] <= '9'; ++i)  /* note condition "is a digit" */
   n = 10*n + (s[i] - '0'); (SLOW)/*  s[i]-'0' is int, add to int 10*n        */
   return n;
}
```

What happens with input 123afg? (stop at a, get integer 123) Computer has to do this sort of thing every time the compiler reads a number you have typed in base 10 as a constant in a program, since computer "sees" a number as an int form, all at once as a single digit!!!

Could just as well type it in base 8 (atoo, ascii to octal). (Ask how to change the program) Or base 16 (atox) (need to special case letter digits). Now jump ahead to program in Section 3.7.

```
/* itoa:  convert int n to characters in base 10 in array s */
void itoa (int n, char s[ ])
{
int i, sign;

if ((sign = n) < 0)       /* record sign */
   n = -n;                /* make n positive */
i = 0;
do  {       /* new loop type -- generate digits in reverse order */
   s[i++] = n % 10 + '0';      /* generate next digit */
   /* what conversion takes place here? */
}  while( n /= 10 > 0);/* delete digit from end of int */
if (sign < 0)
   s[i++] = '-';
s[i] = '\0';
reverse (s);
}
```

The loop do . . . while . . . executes once whether condition true or not. The expression n /= 10 is the same as n = n / 10; Could also be n += 10 same as n = n + 10; Shorthand expression like n++. Note you wrote a reverse function yourself.

Example, with int of -253, generate '3', '5', '2', '-', '\0'. Then reverse.

Note function itoa is not in UNIX Library, but atoi IS in UNIX Library. To achieve aim of itoa would use sprintf:

```
sprintf(s, "%d", n)      /* char s[ ], int n                                    */
```