LECTURE 3: VARIABLES, SYMBOLIC CONSTANTS, FUNCTIONS (K&R §§ 1.2–1.5.1)

A C program consists of variables and functions.

fahrenheit to celsius (version 1)
K&R, page 9

```
#include <stdio.h>

/* print Fahrenheit-Celsius table for
   fahr = 0, 20, ..., 300 */
int main(void) {
  int fahr, celsius;
  int lower, upper, step;

  lower = 0;   /* lower limit of temperature table */
  upper = 300; /* upper limit of temperature table */
  step  = 20;  /* step size */

  fahr = lower;
  while(fahr <= upper) {
    celsius = 5 * (fahr - 32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

C VARIABLES

variable	name given to a storage area that our programs can manipulate lowercase by convention

DECLARATION

declaration	announces the properties of a variable consists of: 1. type name 2. list of variables
type	classification identifying a type of data basic data types include: char, int
All variables must be declared before they are used.	
No memory is allocated.	

DEFINITION

definition	for an:	
	object	causes storage to be reserved for that object
	function	includes the function body
	enumeration constant or typedef name	is the (only) declaration of the identifier

..int upper;
int lower = 0;
	both lines are definitions (because memory is allocated)
	the second also initialized the variable with value 0.

extern int i;	an example of declaration that is not definition (we will cover later)

ASSIGNMENT

The value of this variable can be changed as the program executes.	int lower;
	lower = 0;

TRUE AND FALSE IN C

C does not have a Boolean type (a true or false type).

In c:	true	any numeric value not equal to 0
	false	0

WHILE LOOP

```
while(logical expression) {
  body (statements to execute while true) }
```

the logical expression is tested.

if it is true, the body is executed and then the logical expression is tested again.

when the test becomes false, the loop ends.

note:	no statements are executed if the logical expression is false upon entry

while(fahr <= upper) {
[body]
}
	Here, the loop executes until fahr > upper

fahrenheit to celsius (version 2)
K&R, page 15

```
#include <stdio.h>

#define LOWER 0   /* lower limit of temperature table */
#define UPPER 300 /* upper limit of temperature table */
#define STEP  20  /* step size */

/* print Fahrenheit-Celsius table for fahr = 0, 20, ...,
300 */
int main(void) {
  int fahr;

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

SYMBOLIC CONSTANTS

#define LOWER 0	example of a symbolic constant

A #define line defines a symbolic constant to be a particular string of characters.

why use a symbolic constant?
conveys information about numerical constants.
allows you to easily update them in one place.

Symbolic constant	no memory location assigned to hold value (this is a declaration)
	not an executable statement (notice lack of semicolon at end of line)
	value cannot change during program execution.

Recall from gcc discussion that the preprocessor deals with # commands before actual compilation begins.

#define IDENTIFIER value
The C preprocessor will go through the source code and everywhere it finds IDENTIFIER it will substitute value.

FOR LOOP

```
for (initialization; condition; increment/decrement)
  statement
```

```
for (initialization; condition; modification){
  statement(s)
}
```

initialization	executed once when the loop is started
condition	the loop test statement (when to stop looping)
modification	a statement to execute at the end of each loop (usually an increment or decrement)

example:

```
for(fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
```

initialization	assigns value of symbolic constant LOWER to variable fahr
condition	keep looping as long as fahr does not exceed UPPER
modification	replace the current value of fahr with fahr + STEP

FUNCTIONS

A function is a separate block of code that you can call as part of your function.

A function executes and returns to the next line after you call it in your program.

You can provide a function with arguments inside parentheses following the function name:
function_name(arguments);

Arguments are passed by value (we will talk about this more later).

A return value may be passed back, e.g.:
return_value = function_name(arguments);

printf()

printf is a general-purpose output formatting function.

printf takes a variable number of arguments, instead of a predefined number of arguments.

the first argument in a call to printf determines the total number of arguments the call requires.

note:	in the examples below, □ signifies a space. it is used to illustrate how different printf statements work.

How it works	
In the first argument, you will provide a string of characters that will contain:	
literal characters	these print as they appear in the string e.g., "example□1"
escape characters	used for hard to type string elements e.g., "example□1\nexample□2"
replacement characters	printf will replace these e.g., "example□%3.1f\n"
For each set of replacement characters in the first argument, there must be a corresponding argument following the first argument.	
when printf prints to stdout, it will substitute the replacement characters with the corresponding arguments, using the formatting specified.	

conversion specifications see K&R table B-1 (page 244) for full table			
A format specifier begins with % and indicates the basic formatting of the value that is to be printed with a conversion specifier.		For example: If the 1st format specifier in the 1st argument is %d, then the 2nd argument must be an integer.	
Conversion specifiers for integers:	%d	decimal integer	
	%i		
	%o	octal integer	
	%x, %X	hexadecimal integer	
Conversion specifiers for floating point numbers:	%f	decimal representation	3.1415
	%e, %E	scientific notation	1.86e6 (= 1,860,000)
	%g,%G	use shorter of f or e	3.1 or 1.86e6

you can further specify width and precision:	
width %4d	specifies the minimum number of characters to print. Adds spaces at the front as needed.
precision %.4d %.2f	differs based on conversion specifier. %f: controls # of decimal places shown. %d: controls # of digits shown.
width and precision can be combined, e.g., %6.2f	

```
printf ("Values:□%3d,□%6.1f\n", fahr, (5.0/9.0)*(fahr-32));
```

%3d	says argument 2 must be an integer argument 2 is fahr printf will print fahr using at least 3 spaces
%6.1f	says argument 3 must be a float argument 3 is (5.0/9.0)*(fahr-32) printf will print the value of that expression using at least 6 spaces and with 1 decimal place of precision.

Remember that everything else in argument 1 will be printed literally, so this will produce:

```
Values:□□□0,□□-17.8
Values:□□20,□□□-6.7
[...]
Values:□300,□□148.9
```

shaded characters are literals.

□-17.8 123456	6 spaces used 1 decimal place

REDIRECTION

output redirection	store the output of a process to a file
	$ command > fileName
	sends standard output of command to the file with the name fileName

input redirection	use the contents of a file as input to a process
	$ command < fileName
	executes command using the contents of the file fileName as its standard input.

TYPES

sizes reflect umass system, but may change based on implementation:

char	1 byte capable of holding one character in the local character set Note that chars are stored as bit patterns like everything else. The type char is specifically meant for storing such characters.
int	reflects size of integers on the host machine 4 bytes holds an integer cannot be longer than a long
short	often 2 bytes (must be at least 2 bytes) holds an integer cannot be longer than int
long	8 bytes (must be at least 4 bytes) holds a long integer
long long	8 bytes (must be at least 8 bytes) specified since the C99 version of the standard
float	4 bytes holds a single-precision floating point number
double	8 bytes holds a double-precision floating point number
May be applied to char or any integer type (default is signed):	
signed	include negative numbers, range is: $-\frac{1}{2}(2^n)$ to $\frac{1}{2}(2^n)-1$ (e.g., 8 bits: -128 to 127)
unsigned	always positive or zero, range is from $0-(2^n-1)$ where n is the number of bits in the type

CHARACTER INPUT AND OUTPUT

Standard functions/macros for character input and output are defined in <stdio.h>.

getchar()	
Each time it is called, getchar reads the next input character from a text stream and returns that as its value.	
No arguments are passed to getchar.	

getchar() returns an int value	int c;
	[...]
	c = getchar();

Next character input is returned as value of variable c.	

putchar(c)	
putchar prints a character each time it is called.	
getchar(c)	prints the contents of the integer variable c as a character

COPYING INPUT TO OUTPUT

version 1 (K&R, page 16)

```
pseudocode:
read a character
while (character is not the
       end-of-file indicator)
  output the character just
    read
  read a character
```

```
#include <stdio.h>
main(){
  int c;
  c = getchar();
  while (c != EOF) {
    putchar(c);
    c = getchar();
  }
}
```

We use the int type char for ascii code characters.

We use int c here because we get out of the loop with EOF, which is defined to be not the same as any char value.

EOF	end of file defined in <stdio.h>	ctrl-d

version 2 (K&R, page 17)

```
#include <stdio.h>
main(){
  int c;
  while ((c = getchar()) != EOF) {
    putchar(c);
}
```

Takes advantage of that fact that an assignment is an expression and has a value.	This value is the LHS after assignment.

Note the parentheses around c = getchar(). They are necessary because of precedence rules.