C Programming

- C is the language of choice for systems programming and embedded systems
- You will learn to write, execute, and debug C language programs in this course
- Use Kernighan and Ritchie (K&R) textbook!!

Prof. Duc A. Tran
Department of Computer Science

Disclaimer: Many of these slides are the revised and extended version of those used in earlier offerings of this class (taught by Dr. Ron Cheung and Mr. Glenn Hoffman)

Working Environment

- For grading, I will use the terminal window to compile and test your program.
- In the lectures, I will use **Mac OS** as the operating system, **Xcode** for writing the source code, and **gcc** at the command line as the C compiler
- You can use any OS, text editor, and IDE of your choice to work on your C code, but you need to know how to compile and run your code at the command line in **UNIX**.

Basic UNIX Commands

cat	display a file on your terminal screen (see also “more”)
cd	change directory
cp	copy a file
logout	logout from your account
lpr	print a hard copy
ln	creates a new link to a file
ls	list files in a directory
more	display a file on your terminal screen - one page at a time
mv	move a file from one place to another
mkdir	create a new subdirectory
pwd	print working directory (pathname of directory you’re in)
rm	remove (delete) a file
rmdir	remove (delete) a directory
CTRL-c	“Control” key and “c” key together – stop current command

Visit class website for some basics about UNIX

First Program: Hello World!

- Create and run a C program – “Hello World!” (K&R, p5+)
- Create a source file “**hello.c**” in one of three ways
  - Use a PC in S-3-157, run Putty/SSH and vi or emacs
  - Use your home PC, run Putty/SSH and vi or emac
- Use “**gcc**” to compile and create a file named “hello”
- Run “**hello**” to see the printout on screen
- Run “**script**” to create a “**typescript**” file and run “**exit**” to end the script file

Using **script**

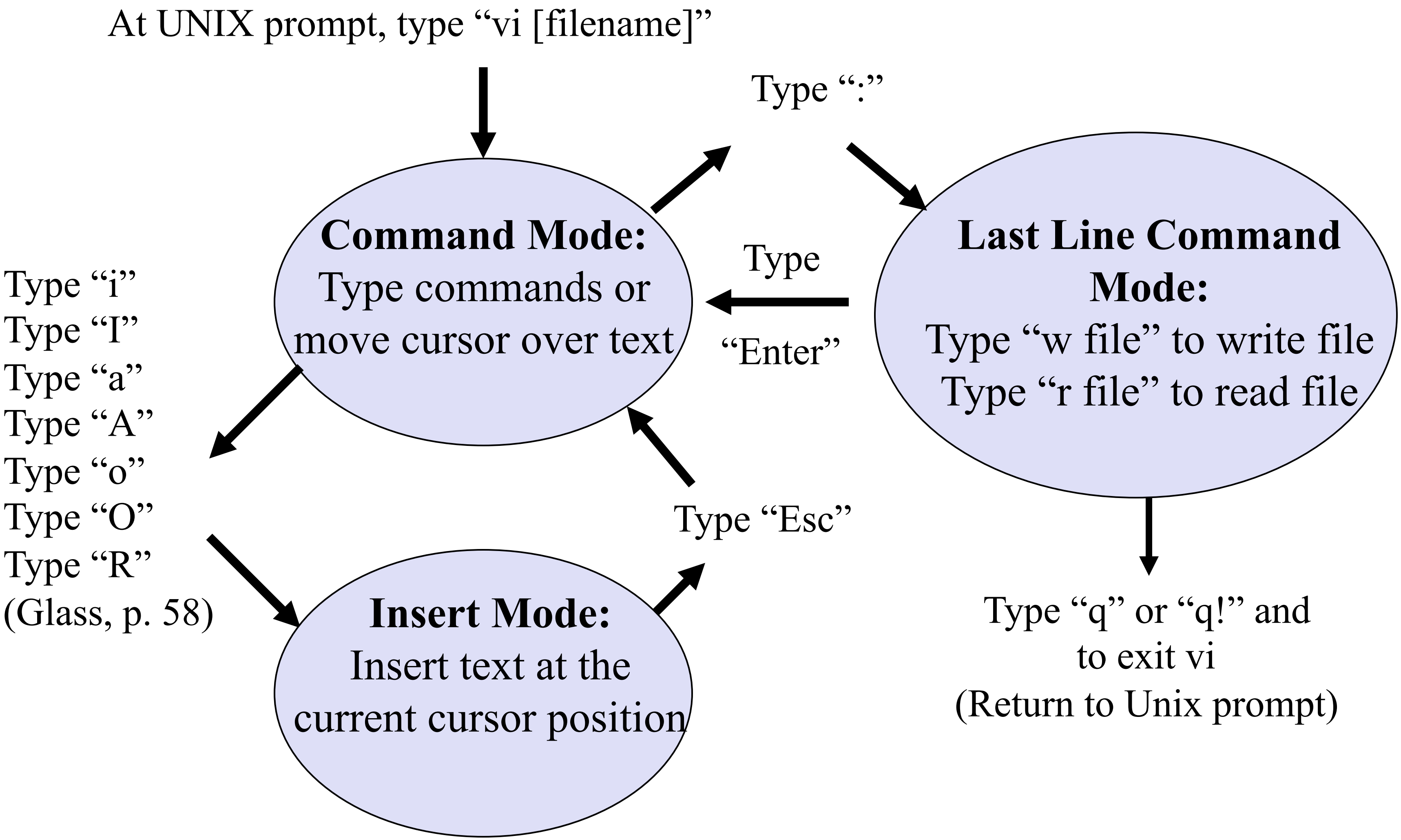
- The "**script**" command: record a terminal session.
- The "**scriptreplay**" command: replay a script.
- The session is captured in a file name "**typescript**" by default to specify a different filename: "**script filename**"

```
% script                                    (Start recording typescript file)
Script started, file is typescript
% ls –l                                     (list directory entries)
% cat hello.c                               (display source file)
% gcc –m32 hello.c –o hello                 (compile source file in 32-bit mode)
% ./hello                                   (run executable from current directory)
% exit                                      (stop recording)
script done on Thu Aug 23 11:30:02 2012
```

Using **vi** or **vim** as Text Editor

- Many like **vi** or **emacs** in Unix as a text editor
  - **vim** is the LINUX version
- Keyboard oriented – no use of a mouse!
- At UNIX prompt, type "**vi hello.c**"
- "**vi**" has three modes (See next slide)
  - "Command mode"
  - "Insert mode"
  - "Last line command mode"

vi Modes


At UNIX prompt, type “vi [filename]”
Type “:”
Command Mode: Type commands or move cursor over text
Last Line Command Mode: Type “w file” to write file Type “r file” to read file
Type “Enter”
Type “i”
Type “I”
Type “a”
Type “A”
Type “o”
Type “O”
Type “R”
(Glass, p. 58)
Type “Esc”
Insert Mode: Insert text at the current cursor position
Type “q” or “q!” and to exit vi (Return to Unix prompt)

vi: Text Entry Commands

- Key Action

- i Text is inserted in front of the cursor
- I Text is inserted at the beginning of the current line
- a Text is added after the cursor
- A Text is added to the end of the current line
- o Text is added after the current line
- O Text is inserted before the current line
- R Text is replaced (overwritten)

vi: Other Commands

- Movement Commands (Glass, page 86)

Up one line	"cursor up" or "k" key
Down one line	"cursor down" or "j" key
Right 1 char	"cursor right" or "l" key
Left 1 char	"cursor left" or "h" key

- Edit commands (Glass, page 87)

[n]x	delete n characters at cursor
[n]dd	delete n lines at current line

- To display line numbers by default, create an .exrc file in your home directory with one line: "set nu". Your new **vi** session should show line numbers.

hello.c Program (K&R, Page 6)

```
/* hello: first program
   name: your name
   date: xx/xx/xx
*/
#include <stdio.h>
int main(void) {
   printf("Hello World!\n");
   return 0;
}
```

comment

C preprocessor directive

C function

statements

Comment Lines

- Comment text is ignored by the compiler

  /* This is a multi-line comment.
  Write whatever you want here
  The compiler ignores all these lines. */

- Be sure to start with /* and close with */

Include a Library - **#include** …

- Because this program uses the **Standard I/O Library**, it needs to include <stdio.h>

- In C programming, a "**.h file**" defines
  - Macros (e.g. Names for constants)
  - Prototypes for functions (e.g. **printf** itself)

- "**gcc** won't compile "**hello.c**" with the "**printf**" function without the "**#include <stdio.h>**"

Main Function

- "**int main (void)**" is where your C program starts execution
- Every function start with { and close with }. The code to implement this function put between these "braces"

```
{
        program statements are here;
}
```

printf

- The Standard I/O Library provides a function named "**printf** (…)" to display argument as text on screen

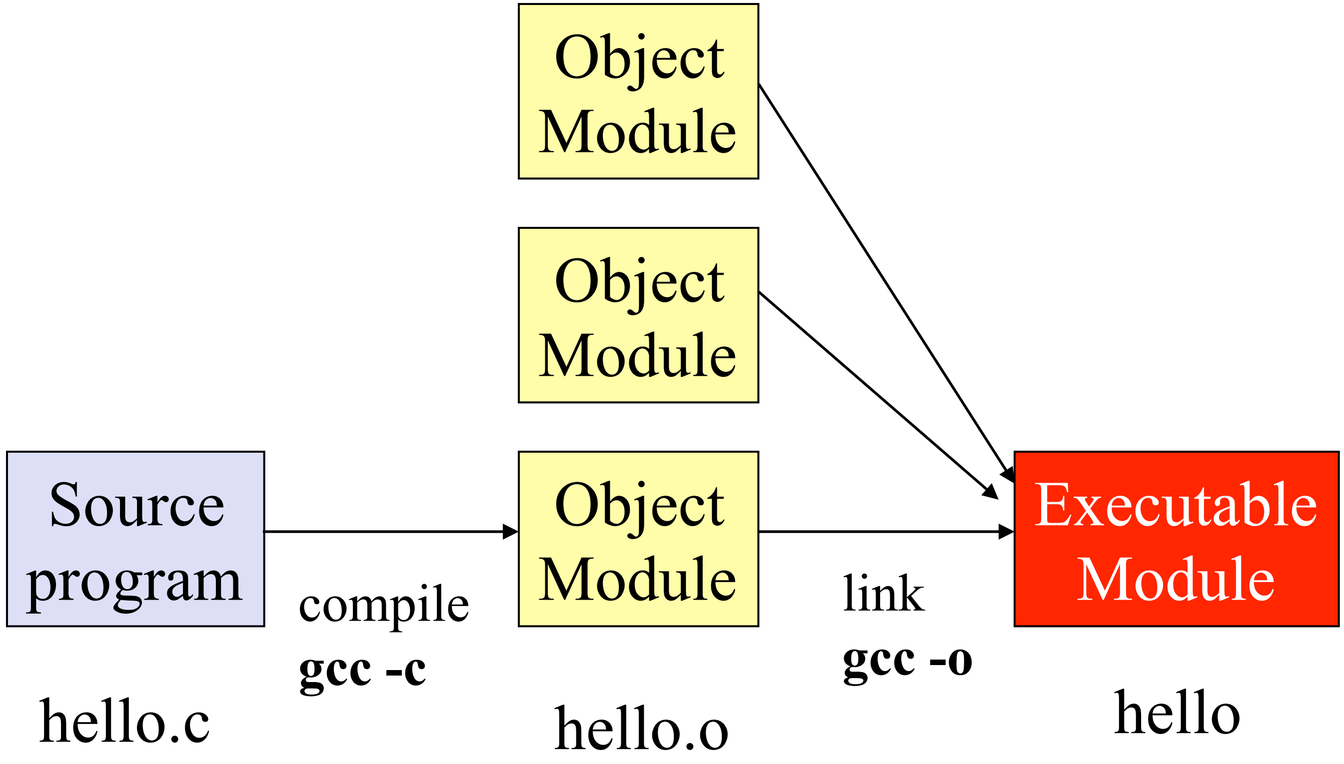
printf("Hello World!\n");

- "**\n**" is a C convention for "end of line" (character constants in K&R page 193)
- All C program statements end with a " **;** "

Character Constants

New line	\n	backlash	\\
Horizontal tab	\t	question mark	\?
Vertical tab	\v	single quote	\’
Backspace	\b	double quote	\”
Carriage ret	\r	octal number	\ooo
Form feed	\f	hex number	\xhh
Audible alert	\a		

Compiling and Linking



Compile Your Program

- To compile your program, type
-

  gcc hello.c –o hello

  - To build a 32-bit application: **gcc –m32 hello.c –o hello**
- If you get no error messages
  - The compiler has accepted your source code
  - You should now have a file named "**hello**"
  - If you forget to specify **–o hello** in "**gcc hello.c**", the default executable will be a file name "**a.out**"

Run Your Program

- At UNIX/LINUX prompt, type **./hello**
- If you get the printout "Hello World!" and a new prompt, your program ran successfully
- If not,
  - Study any UNIX error messages for clues
  - Study your source code for logical errors
  - Probably logical errors - compiler didn't catch
  - Fix your source code and recompile / rerun

Debugging a C program error

- There is a big difference between:
  - The program compiling correctly
  - The program doing what you want it to do
- You hope the compiler will catch your errors
  - These errors will be easier to find
- If the compiler does not catch your errors
  - These errors will be harder to find

Compiler Error Messages

- A compiler error message may direct you to a specific error in your program
- A compiler error message may be vague about what the error is and why it is an error
- Some compilers are better than others at providing useful error messages!

Compiler Error Messages

```
#include <stdio.h>
int main(void)  {
    printf("Hello, World!");
    return 0;
    /*  missing “}”  */
```

% gcc hello.c –o hello
hello.c: In function `main':
hello.c:6: parse error at end of input

- Not a very helpful message!

Variables

- Defined Data Type, Name, and (= value)

  int lower = 0; /* Note: "=" and ";" */

- lower case by convention for readability
- An executable statement
- Memory location assigned to hold the value
- Value can be changed as program executes

  lower = 20; /* Legal */

Symbolic Constants

- Defined Name and Value

  #define LOWER 0 /* Note: No "=" or ";" */

- UPPER CASE by convention for readability
- Not an executable statement
- No memory location assigned to hold value (known as declarations)
- Value can't be changed as program executes

  LOWER = 20; /* NOT Legal */

Example Program (K&R, P 15)

declarations

definition

statements

```
#include <stdio.h>
#define LOWER 0      /* Symbolic Constants */
#define UPPER 300
#define STEP 20
/* Print out Fahrenheit – Celsius Conversion Table  */
int main() {
    int f;                  /* Variable type*/
    for (f= LOWER; f<= UPPER; f = f + STEP)
      printf("%3d,%6.1f\n", f, (5.0/9.0)*(f– 32));
    return 0;
}
```

for Statement

for (A; B; C) – repeat executing statement(s) within the loop

A is initialization (executed once when loop is started)

B is the loop test statement (when to stop looping)

C is a statement to execute at end of each loop

Example

```
for (f= LOWER; f<=UPPER; f= f+ STEP) {
   statements within the loop;
}
```

printf statement (K&R, p. 154)

printf (“%3d, %6.1f\n”, f, (5.0/9.0)* (f- 32));

First argument = “**%3d, %6.1f\n**”

%3d = integer format with 3 digits

%6.1f = floating point format with 6 digits and 1 decimal

n = end of line character just as in “Hello World!”

Second argument = **f**

Third argument = **(5.0/9.0)*(f– 32.0)**

printf formats

printf (“%3d, %6.1f\n”, f, (5.0/9.0)* (f- 32));

- **%3d** and **%6.1f** are special placeholders
- The two expressions following the quoted string, **f**, and **(5.0/9.0)*(f-32)**, are to be printed according to the prescription given, respectively.
- Other characters in the quoted strings are printed verbatim

Function

- A function is a separate block of code that you can call as part of your program
- A function executes and returns to next line after you call it in your program
- Arguments may be passed to a function
- Arguments are passed by value

  function_name (arguments);
- A return value may be passed back

  return_value = function_name (arguments);

Character I/O – **getchar**( )

- A standard function/macro defined in <stdio.h>
- Get a **int** value representing a character from standard input
  - No argument needed

```
int c;
c = getchar( );
```

Character I/O – **putchar**( )

- A standard function/macro defined in <stdio.h>
- Print the character to standard output
- Argument: the **int** value representing the character from standard input

```
int c;
putchar( c);
```

int vs. **char**

- **int** is an integer type, 4 bytes of significance, from -2^31 to 2^31 -1.
- **char** is another integer type, but only 1 byte of significance from -128 to 127

- What is a character? (‘a’, ‘b’, ‘1’, ‘2’, etc.): Values in the range of 0-127 decimal are ASCII code characters.
  - These characters each fits in 1 byte.
  - Therefore, we should use type **char** to represent a character

ASCII Code

- For computers to process our letters, digits, punctuation marks, etc, we need a binary code for each such “character”.
- American Standard Code for Information Interchange (ASCII) provides these codes.
  - See the ASCII Code Table on the next slide
- Standard 8 bit bytes and 16 bit words are not integer multiples of 3 bits but are integer multiples of 4 bits – favoring use of Hex!

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	&#127;	DEL

Source: www.asciitable.com

Octal and Hex Numbers

- People normally deal in numbers base 10
- Computers normally deal in numbers base 2
- The problem:
  - Reading a long string of 1's and 0's not easy
  - Conversion between base 2 and base 10 not easy
- The solution:
  - Convert binary digit strings to Octal or Hex
  - Easily done because $2^3 = 8$ and $2^4 = 16$

Octal and Hex Numbers

- Look at a long string of binary digits in groups
  - 3 digits for Octal
  - 4 digits for Hex
- See the following examples:

– Binary Digits	011010101100 …
– Grouped by threes	011 010 101 100 …
– For Octal	003 002 005 004 …
– Grouped by fours	0110 1010 1100 …
– For Hex	0x6 0xa 0xc …

- Don't convert binary to/from Hex/Octal via decimal!

“Octal” Dump

- Use “od –x” to see hex dump of a file

  od –x trim.in

  00000000 0909 4e68 …. 2020

  . . .

  00000120 7061 7274 …. 0a0a
- Octal and Hexadecimal numbers
- Why dump in Hex instead of Octal?
- ASCII code for representing characters

Example: File Copying

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

This program takes whatever you get from standard input (keyboard) and prints it out at standard output (screen)

EOF: a special **int** constant representing the end of file (in this case, end of standard input)

Here, variable c means a character. Why do we define it as an **int?**

Redirecting **stdin** and **stdout**

- We can use the previous program, **filecopy**, to copy a file into another. How?
  - Redirect **getchar**( ) to read from a file, instead the standard input (stdin)

    filecopy < input.txt

  - Redirect **putchar**( ) to write to a file, instead the standard output (stdout)

    filecopy > output.txt

Example: Counting Lines

```
/* linecount.c */
#include <stdio.h>
main ( ) {
   int c, m;
   m = 0;
   c = getchar();
   while (c!= EOF) {
         if (c= = '\n' ) ++m;
         c=getchar( );
   }
   printf("%d\n", m);
}
```

Stop the loop when we see EOF (end of file)

Each time we see the new line character '\n', we increment the count m

Check for Equality

- Use double equals (= =) for checking "equals"
- if (c = = '\n')
  - If statement with logical expression in parentheses
    - Result of comparison equal to 0 is treated as False
    - Result of comparison not equal to 0 is treated as True
  - The expression is a check for int c equal to '\n' or not
- if (c = '\n')
  - If int c wasn't equal to '\n' before, it is now!
  - And the expression is treated as true ('\n' is not = 0)

Increment, Decrement

- Incrementing a variable

  Shorthand ++m;

  Shorthand m++;

  Equivalent to m = m + 1;

Prefix: increment m before m is used

Postfix: increment m after m is used

- Decrementing a variable

  Shorthand --m

  Shorthand m--

  Equivalent to m = m - 1

The **while** loop

while (logical expression) {

statements while expression is true;

}

- **while** does not execute any statements if the logical expression is false upon entry!

The **for** loop

for (initialize; loop test; increment) {

statements for expression is true;

}

- **for** does not execute any statements if the loop test is false after initialization!

The **if-else** statement

if (logical expression) {

statements when expression is true;

} **else** {

statements when expression is false;

}

- "**else**" portion of statement is optional!

Nested **if-else**

if (logical expression 1) {

statements when expression is true;

} **else if** (logical expression 2) {

statements when expression is false;

} **else if** (logical expression 3) ….

- Inside a if-else statement block, we can have other if-else statements

Array / Character String

- An array is a list of a given number of values of a given type. The name of the array is a pointer to the memory space where its elements are stored

  int array[100];

- Character string = is an array of **char** type values ending with a null character ('\0')

  char name[50];

Arrays / Character Strings

- How to use a variable to store the string "hello\n"?

char array[7] = "hello\n";

- Make sure that the last element's value is '\0'
- The values of this array are

array[0]	array[1]	array[2]	array[3]	array[4]	array[5]	array[6]
'h'	'e'	'l'	'l'	'o'	'\n'	'\0'

Example: Counting Digits

```
/* count.c */
/* count digit characters 0-9 coming from stdin */

#include <stdio.h>
int main() {
    int c, i;     /* c for getchar - ASCII code for integers */
    int ndigit[10];    /* subscripts 0 through 9 */
    for (i = 0; i <= 9; ++i)     /* Set all array value = 0 */
      ndigit[i] = 0;
```

Example: Counting Digits, cont’d

```
    while ((c = getchar()) != EOF) {
        if(c >= '0' && c <= '9')  /* if c is a digit */
            ++ndigit[c-'0'];      /* increment 1 array element */
    }
    printf("digits = ");
    for (i = 0; i <= 9; ++i) printf("%d ", ndigit[i]);
    printf("\n");
    return 0;
}
```

Run count.c

% gcc count.c

% ./a.out

12345678901122233334444455555567778889999000

fgfgfgfg (Note: These won't be counted as digits)

^D (Control-D is End of File – EOF)

digits = 4 3 4 5 6 7 2 3 4 4

%

Example: maxline.c

- Find the longest line. Here is the pseudocode:

```
while (there’s another line)
        if (longer than the previous longest)
                save it
                save its length
print longest line
```

- Large enough to break up into “functions”

maxline.c

```
#include <stdio.h>

/*  define maximum length of lines */
#define MAXLINE 1000

/*  define function prototypes  */
int getline(char line[], int maxline);
void copy(char to[], char from[]);
```

Program: maxline (cont' d)

```
int main ( ){
    int len, max=0;                          /* initialization */
    char line[MAXLINE], longest[MAXLINE];
    while ((len = getline(line, MAXLINE)) >0)
        if (len > max) {
                max = len;
                copy(longest, line);
        }
    if (max > 0) printf (“%s”, longest); /* there was a line  */
    return 0;
}
```

Function: getline( )

```
/* 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;
    if (c = = ‘\n’) {
        s[i] = c;
        ++i;
    }
    s[i] = ‘\0’;
    return i;
}
```

Function: copy ( )

```
/* copy:  copy 'from' into 'to'
   assume size of array 'to' is large enough  */

void copy (char to[], char from[])
{
   int i;
   i = 0;
   while ((to[i] = from[i]) != '\0' )
         ++i;
}
```

an array of characters; length unspecified

Notes on the Details

- Precedence of operators in getline( )

```
i < lim-1;
((c = getchar()) != EOF);
(expression && expression && expression)
```

- Pass by value arguments for copy (pointers)

```
void copy(char to[], char from[])
while ((to[i] = from [i]) != '\0' )
```

Debugging

- 2 ways to debug a program:
  - Use **printfs**
    - Insert printf's in multiple places in your program and print out intermediate values
  - Use **gdb** debugger
    - A professional programmer uses a debugger, rather than putting in lots of printf statements to track down a bug.
- Most IDEs provide a debugger tool that is much easier to use than gdb at the command line
- But gdb is good if we want to program at the low level

Use of the gdb Debugger

- Start with the correct compiler options:

  gcc -g vt.c -o vt

  creates an executable that has debugging info, e.g.
  - data type for variables/functions
  - correspondence between line # and addresses

- Type the following to run the program:

  gdb vt

- Gives message:

  Ready to run -- not yet running.

Use of the **gdb**

- Want to interact with running program, not letting it run free. To set a break point at main(), type:

  b main

  break at main()

- To run, type:

  r <vt.in

  run, taking stdin from vt.in

- Will stop when encounters main() in program execution -- often lot of things get done first.
- Now can single step through program, s or n (skip entering functions), put out values of variables.

Examples of **gdb** commands

p i	(print value of variable i)
p i=2	(set the variable i to 2 and print it)
p 3*i	(print value of expression 3*i)
p/x i	(print in hex format value of variable i)
set variable i=5	(set the variable i to 5 without printing)
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)

Use of **gdb** (cont'd)

- More complex gdb commands in User's Guide.
- Setting breaks/conditional breaks at line numbers:

  b 36

  b fn.c:22 if i = = 3

- Getting line numbers from "list" or "l" command:

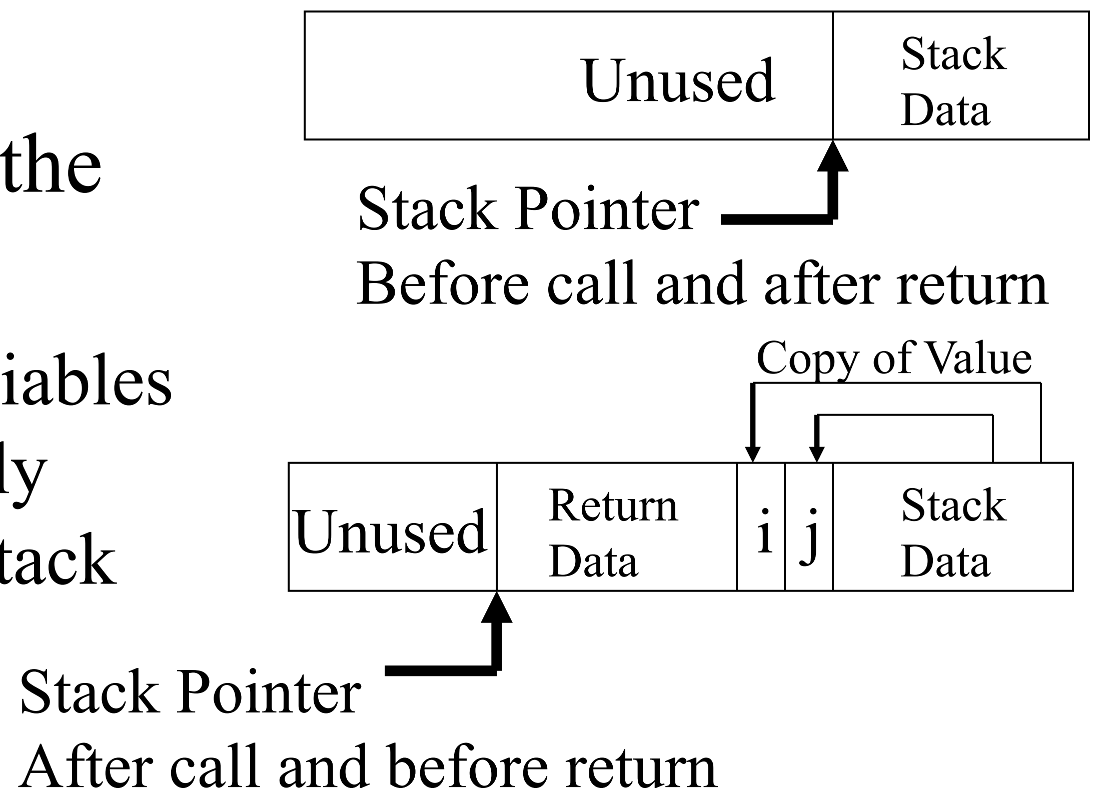
l 22	print 10 lines around line 22 in main
l	after listing some lines, then l means next 10 lines
i b	to get info on breakpoints
d 3	to delete bkpt 3
c	for continue after bkpt encountered

Function: Call by Value

```
void foo(int i, int j) {
}
foo(i, j);
```

Note: Stack pointer is a register

- Pass values as arguments into the function
  - The passed variables are actually only <u>copies</u> on the stack



Function: Call by Value

```
void foo(int i, int j) {
}

foo(i, j);
```

- This is known as Call by Value.
- You can't change arguments in original location within the function -- just change the stack copy
- To make changes, you must pass pointers to original variables. See next slide.

The following doesn't work!!!

```
void exchgint (int a, int b) {
    int dummy;
    dummy = a;
    a = b;
    b = dummy;
}
```

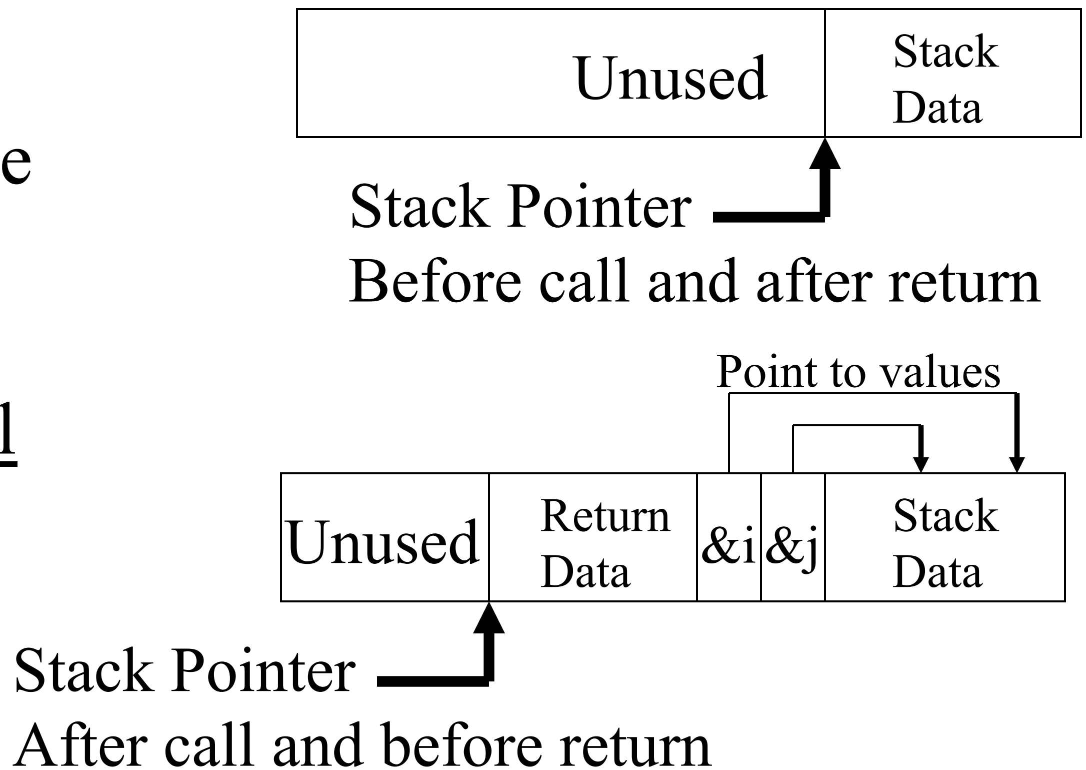
```
int a = 4;
int b = 5;
exchgin(a, b);
/* still, a=4, b=5 */
```

Outside, let's say **a=5, b=4**, and we call **exchgin(a, b)**, then the values of **a** and **b** won't swap.

Function: Call by Reference

- Pass **pointers** as arguments into the function
  - Still only value on the stack <u>but we can access original location indirectly</u>

```
foo(&i, &j);
void foo(int *i, int *j) {
}
```



What is a Pointer?

- Pointer = a variable that represents a memory address
- For example

  int* pi;

  char* pc;

  float* pf;

  - **pi** is a **pointer,** representing a memory address where an integer is stored
  - **pc** is a **pointer,** representing a memory address where a character is stored

Pointers as Arguments

- Must be done with pointers!!!

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

```
int a = 4;
int b=5;
exchgin(&a, &b);
/* now, a=5, b=4 */
```

- **int *** is a **pointer** type. A variable of this type (e.g., **pa**) is to represent a memory address
- Expression ***pa** represents the value stored at the address **pa**

&a is the the address (pointer) where variable **a** is stored. Here, we pass arguments into the function by **pointers** (**&a** and **&b**)

An Array as a Pointer

```
int array1[10], array2[10];
foo(array1, array2);
```

- When passing an array, it is automatically passed as a pointer
- You don't need to create a pointer yourself with the "address of" operator (&)
- This is because by convention, the array variable **array1** is the address where the array begins. It is therefore a pointer.

Local Automatic Variables

- Local variable = defined inside a function (or block{ }), valid only inside this function.
- Local variables are said to be automatic
  - Automatically created when function is called and go away when function is finished
- Memory is allocated on the stack after the calling
- Undefined (i.e. garbage) value unless explicitly initialized in the source code
- Initialization is done each time the function or block is entered

Local **static** Variables: Example

```c
#include <stdio.h>

void increment() {
	static int i = 5;
	printf("%d\n", i);
	i++;
}
```

```c
int main() {
	increment();
	increment();
	increment();

	return 0;
}
```

Each time, **increment**() is called, local static variable **i** value is preserved for future use

Local **static** Variables

- A **static** variable declared in a function is preserved in memory. Local, only used inside { }.
- Set to zero if it is not initialized otherwise.
- Initialization is done only **once** and when the program **starts** execution (K&R P.85).
  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 in the beginning and
    ...                      remember value between calls to rand */
}
```

External Variables

- External variable = defined outside every function (or block{ }), usable everywhere (even in a different file, for example, of a project).
- Don't use them. Why?
  - If their value is corrupted, NOT easy to figure out
  - They make the functions depend on their external environment instead of being able to stand alone using arguments to get their input values and a return value to pass back an output value.
- Software architecture/design standards for most projects will prohibit use of "global variables" or severely restrict their use.

External Variables: Example

```c
/* file1.c */
int i;
extern void f();
int main() {
        f();
        printf("%d\n", i);
        return 0;
}
```

```c
/* file2.c */
extern int i;
void f() {
        i++;
}
```

A project with 2 programs. The external variable **i** in **file1.c** can be used everywhere in the project (**file2.c**)

External Variables: **extern**

```c
/* file1.c */
int i;
extern void f();
int main() {
        f();
        printf("%d\n", i);
        return 0;
}
```

```c
/* file2.c */
extern int i;
void f() {
        i++;
}
```

The external variable **i** in **file1.c** is declared as "**extern**" in **file2.c** so that it can be used in **file2.c**. is also applicable to functions.

Global **static** Variables

- To limit the scope of a global variable to this file only, declare it as **static**
- Can be used to pass data between functions in file only
- Values are preserved like static local variables
- It is guaranteed to be initialized to zero
- If initialized, it is done once before the program starts execution.
- These are more acceptable than external (non-static) variables

Examples of Scopes of Variables

- These examples are from p.342:

"C Programming for Scientists and Engineers with Applications" by Rama Reddy and Carol Ziegler, Jones and Bartlett 2010.

Scope of Variables – Example #1

```
#include <stdio.h>
int main() {
  int x=10;
  printf(“x=%d\n”, x);
  {
     int x=5;
     printf(“x=%d\n”, x);
     {
         int x=40;
         printf(“x=%d\n”,x);
     }
       x=x+4;
       printf(“x=%d\n”,x);
  }
  x=x+15;
  printf(“x=%d\n”,x);
  return 0;
}
```

10

5

x=40

x=5+4

x=10+15

If the same variable is defined
inside and outside the block,
the name inside the block will be referenced if the block is being executed.

Scope of Variables – Example #2

```c
#include <stdio.h>
void func1(void);
void func2(void);
void func3(void);
int main() {
  int x=20;
  printf("x=%d\n", x);
  func1();
  x=x+10;
  printf("x=%d\n", x);
  func2();
  x=x+40;
  printf("x=%d\n",x);
  func3();
  return 0;
}
```

20

30

70

Scope of 1^{st} x

```c
int x;
void func1(void){
  x=5;
  printf("In func1 x=%d\n",x);
  return;
}

void func2(void){
  int x=0;
  printf("In func2 x=%d\n", x);
  return;
}

void func3(void){
  printf("In func3 x=%d\n", x);
  return ;
}
```

5

0

5

Scope of 3^{rd} x

Scope of 2^{nd} x

Scope of Variables – Example #3

```
#include <stdio.h>
void func1(void);
void func2(void);
int main(){
  extern int x;
  x=1;
  printf("x=%d\n", x);      // 1
  func1();
  x=x+6;
  printf("x=%d\n", x);      // 11
  func2();
  x=x+7;
  printf("x=%d\n",x);       // 28
  return 0;
}
```

File 1

Scope of **x**

```
int x;
void func1(void){
 printf("In func1 x=%d\n",x);      // 1
 x=5;
}
void func2(void){
  x=x+10;
  printf("In func2 x=%d\n", x);    // 21
}
```

File 2

Scope of Variables – Example #4

File 1

```c
#include <stdio.h>
void func1(void);
void func2(void);
int main(){
  extern int x;
  x=1;
  printf("x=%d\n", x);
  func1();
  x=x+6;
  printf("x=%d\n", x);
  func2();
  x=x+7;
  printf("x=%d\n", x);
  return 0;
}
```

1

11

18

File 2

```c
int x;
void func1(void){
   x=5;
   printf("In func1 x=%d\n",x);
}
void func2(void){
  int x=10;
  printf("In func2 x=%d\n", x);
}
```

5

10

Scope of x

Scope of x