

# Programming in C



## Chapter 1

### Introduction to C

```
/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;
    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}

#include <stdio.h>

#define IN 1 /* inside a word */
#define OUT 0 /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;
```

```
/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;
    state = OUT;
    nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}

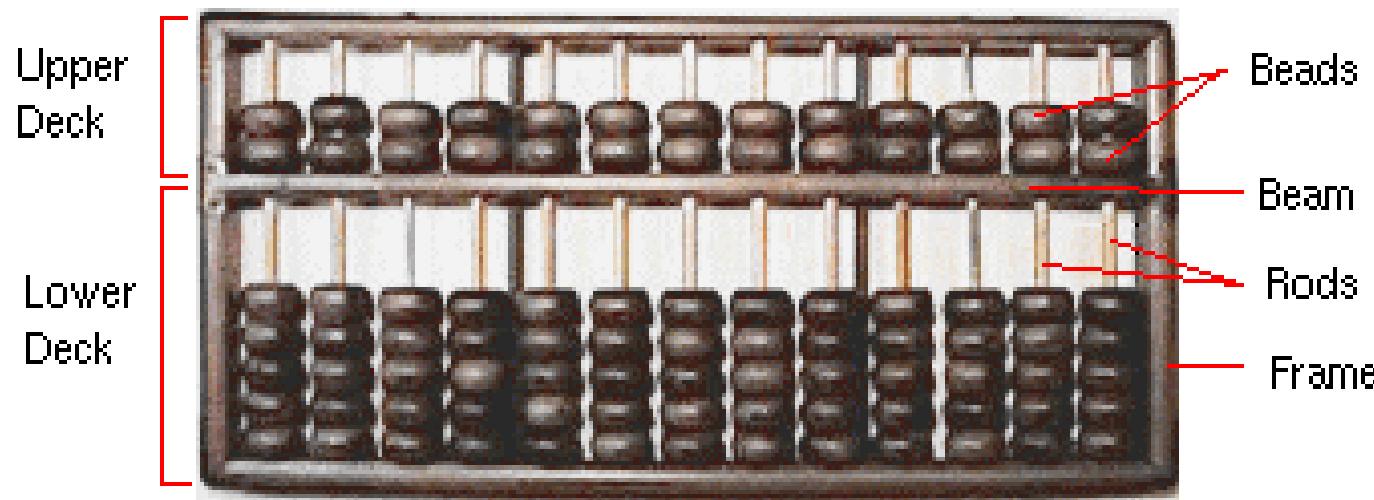
#include <stdio.h>

#define IN 1 /* inside a word */
#define OUT 0 /* outside a word */

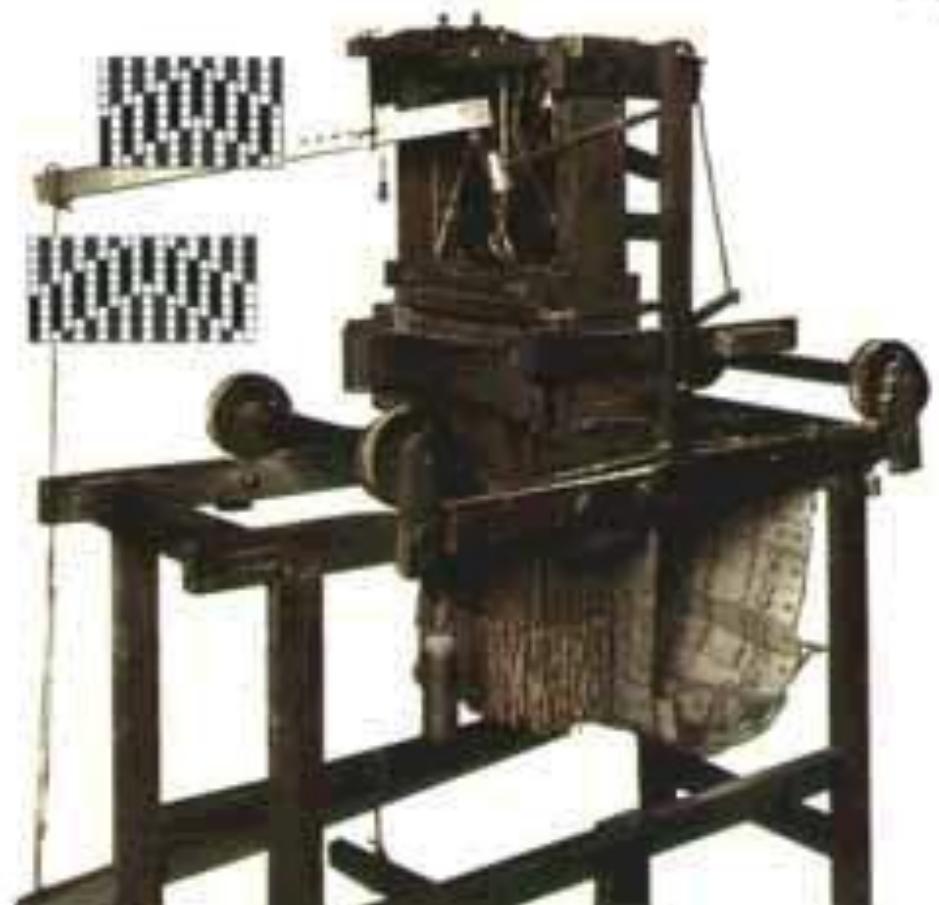
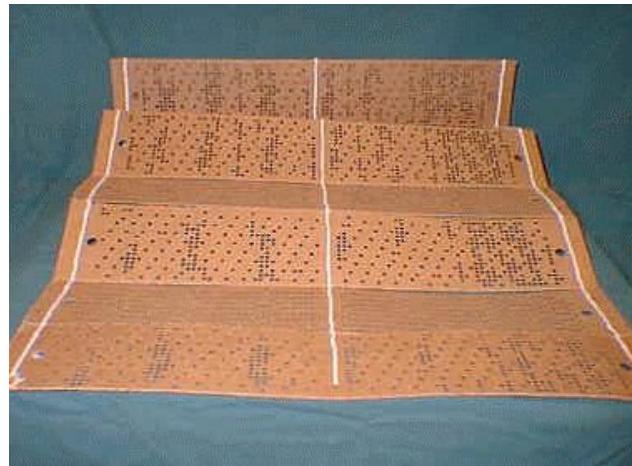
/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;
```

# The Abacus

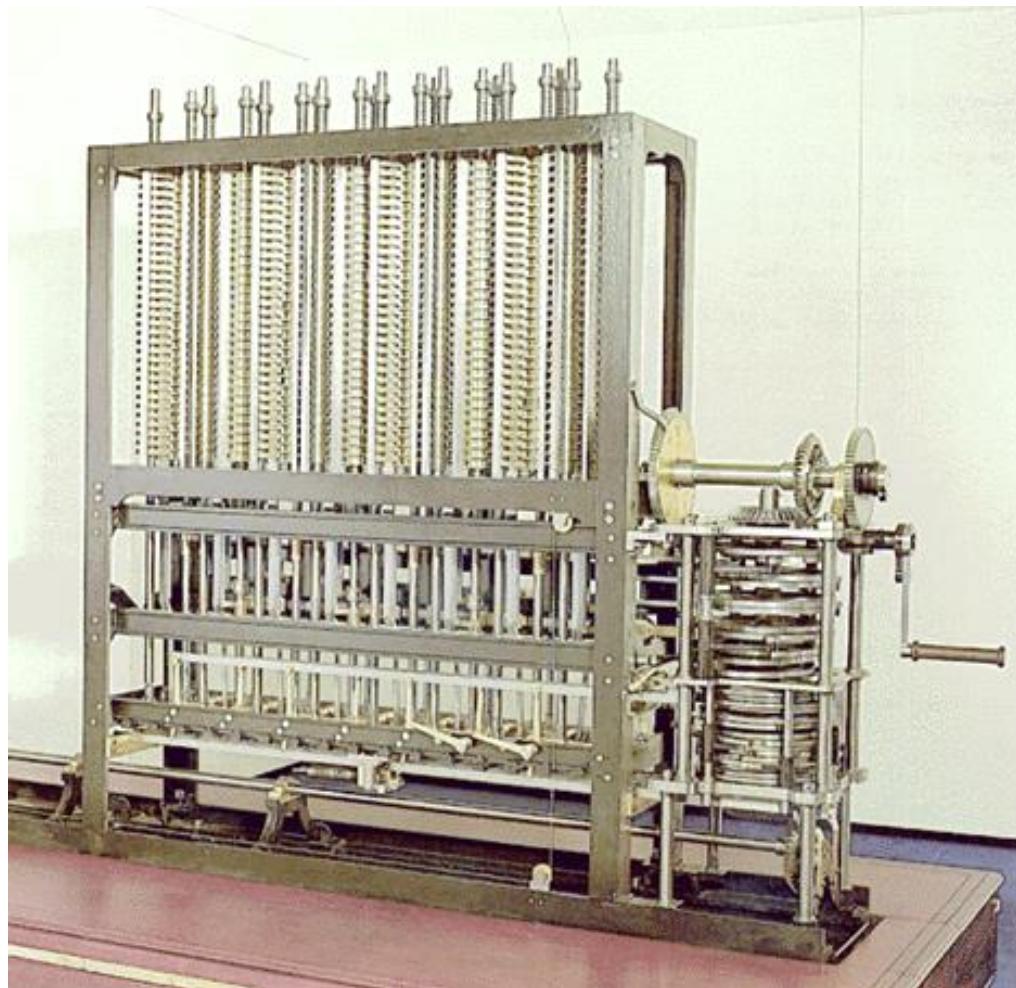
- The abacus, a simple counting aid, may have been invented in Babylonia (now Iraq) in the fourth century B.C.



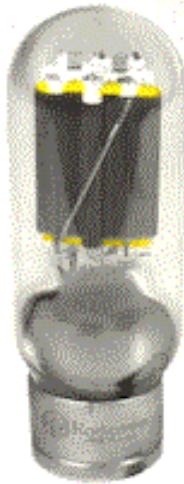
# Jacquard Loom



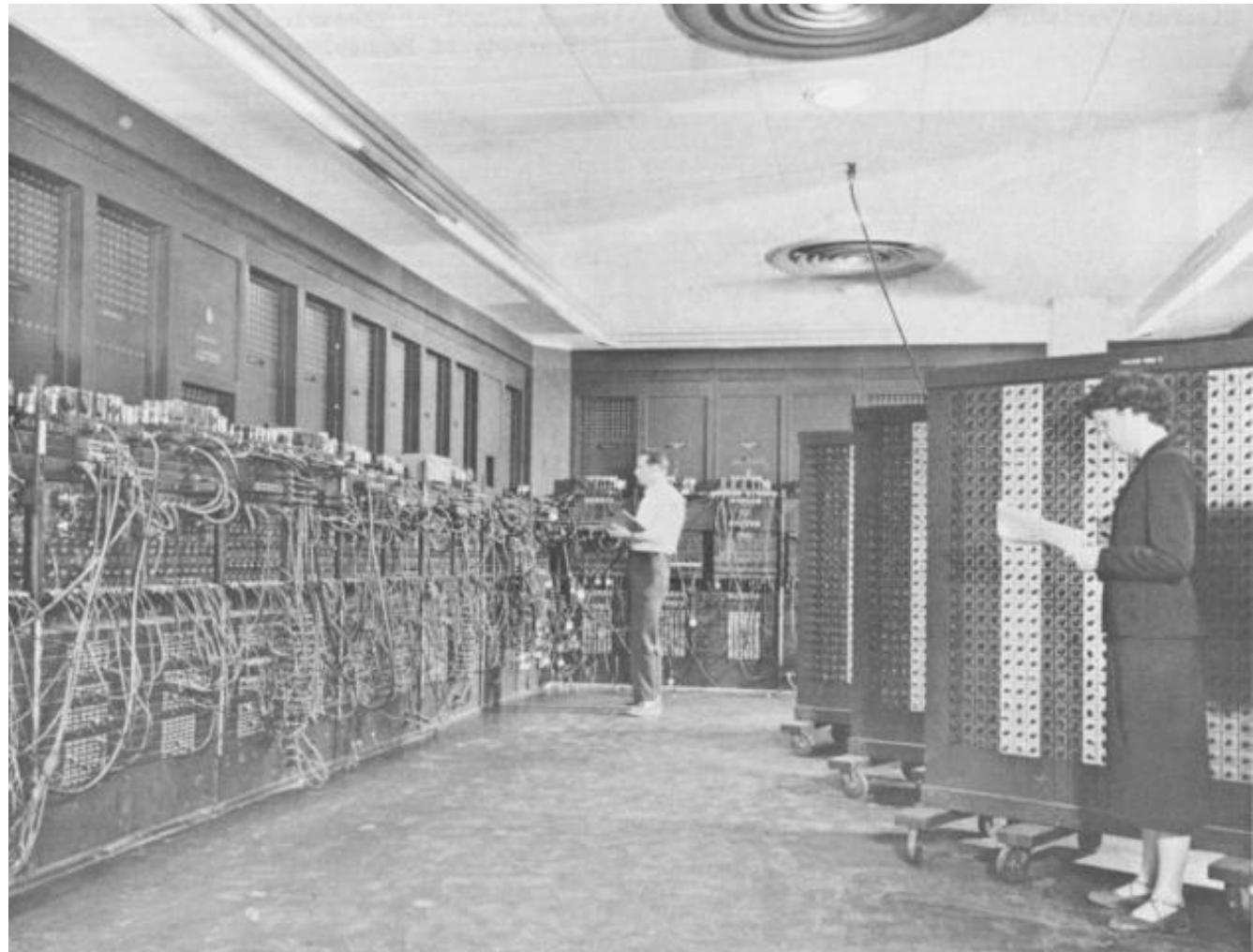
# Babbage Difference Engine, reconstructed by the British Government in 1991.



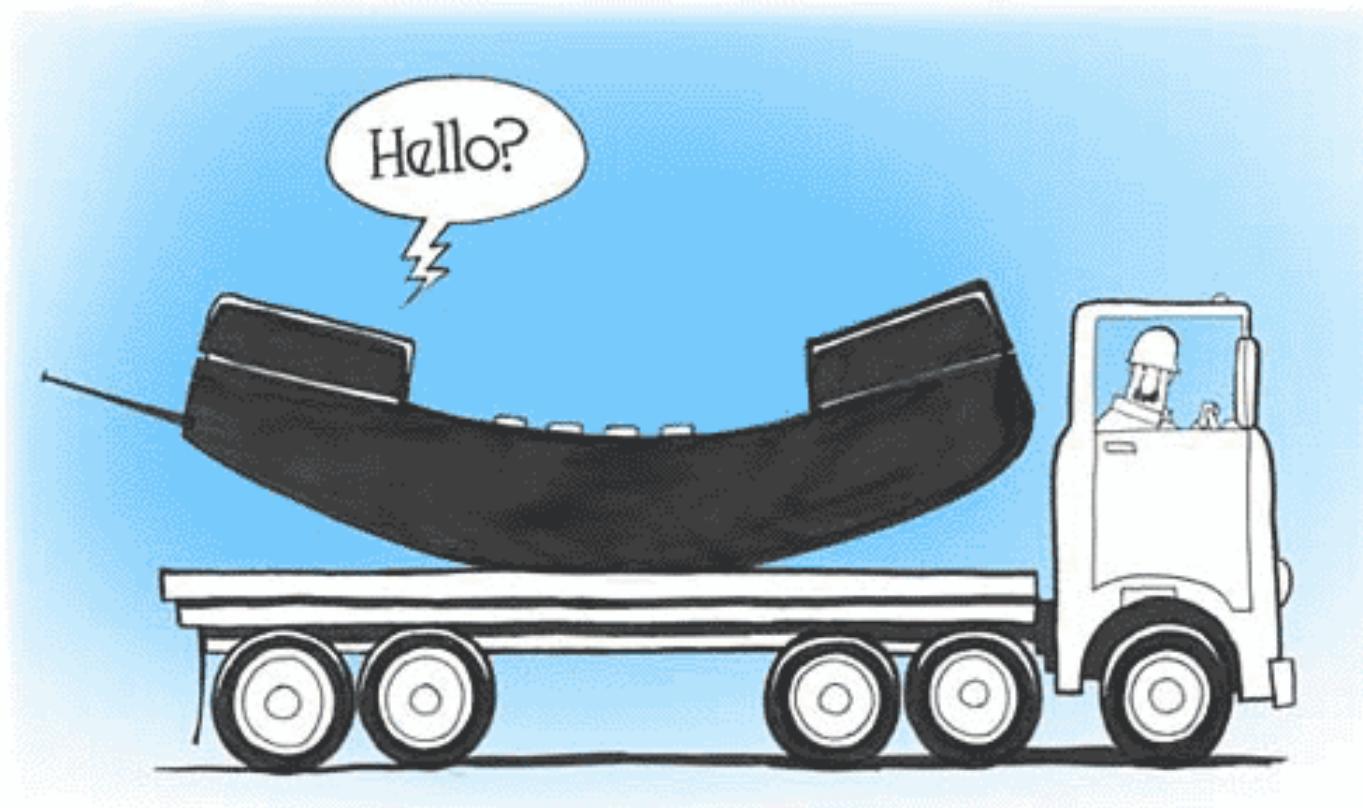
# The ENIAC



Vacuum  
Tube



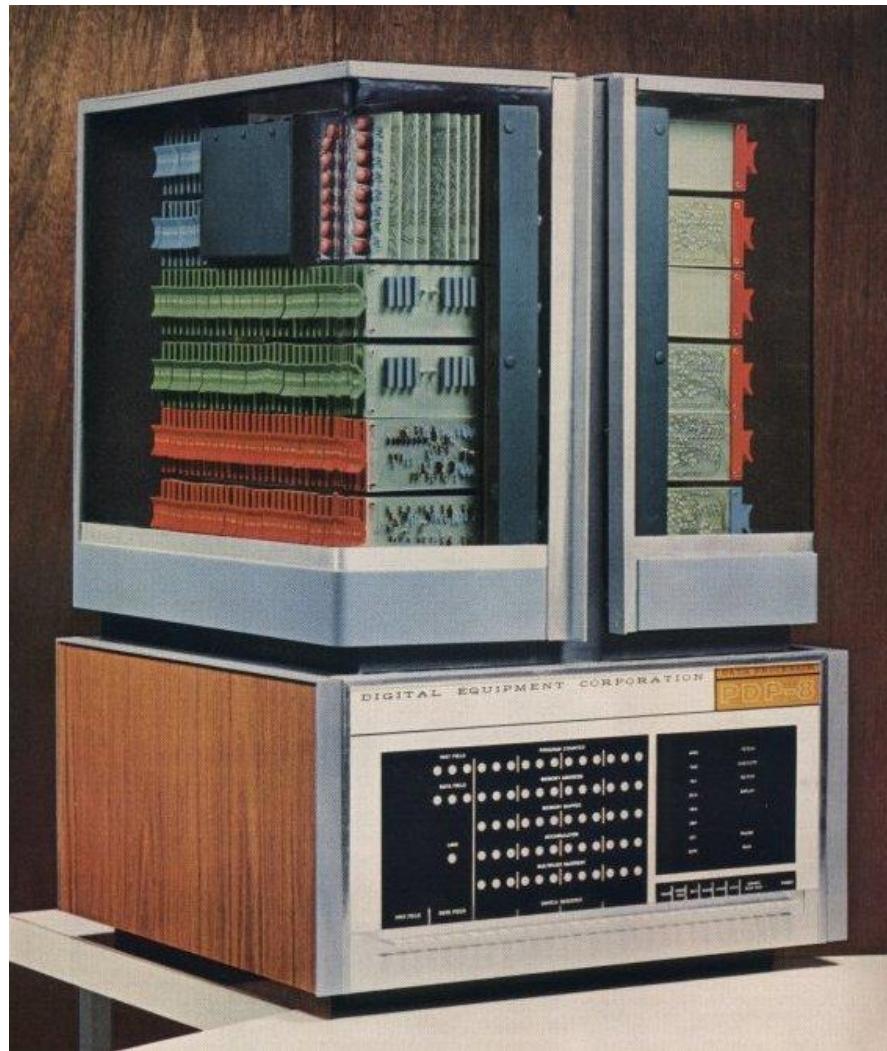
# The size of a cell phone built with Vacuum Tubes



# The IBM 360

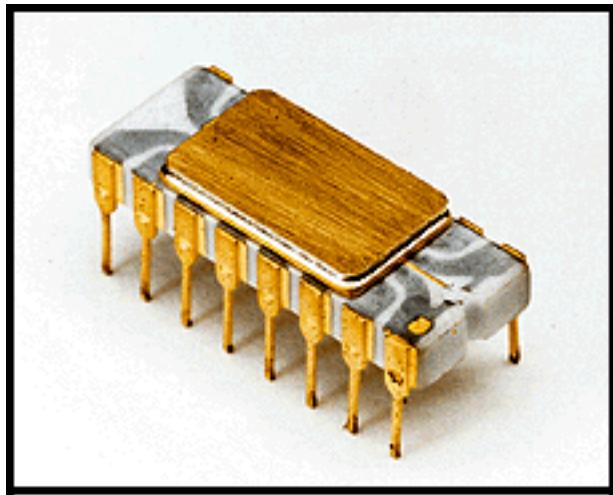


# The PDP-8



# The Microprocessor

- A computer chip that contains on it the entire CPU
  - Mass produced at a very low price
  - Computers become smaller and cheaper
- Intel 4004 – the first computer on a chip, more powerful than the original ENIAC.
- Intel 8088 – used in IBM PC

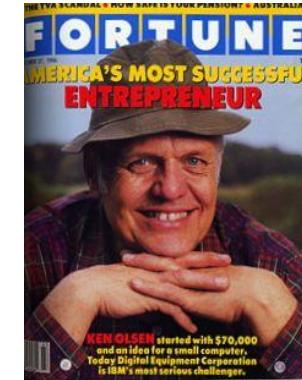


**The Intel 4004, it was supposed to be the brains of a calculator. Instead, it turned into a general-purpose microprocessor as powerful as ENIAC.**



# Famous Quotes about Computers

- “I think there is a world market for maybe five computers.” – Thomas Watson, chairman of IBM, 1943
- “There is no reason anyone in the right state of mind will want a computer in their home.” – Ken Olson, President of Digital Equipment Corp, 1977.



# Hardware

- ***Hardware*** – the physical devices that make up a computer (often referred to as the computer system)



# Hardware Core



- CPU (Central Processing Unit)
  - **CPU (machine) cycle** – retrieve, decode, and execute instruction, then return result to RAM if necessary
  - CPU speed measured in gigahertz (GHz)
    - **GHz** – number of billions of CPU cycles per second
- RAM (Random Access Memory)
  - Also called Memory, Main Memory, or Primary Storage
  - Measured in gigabytes (GB, billions of bytes) today
    - Byte → Character
  - RAM is volatile
    - Temporary storage for instructions and data

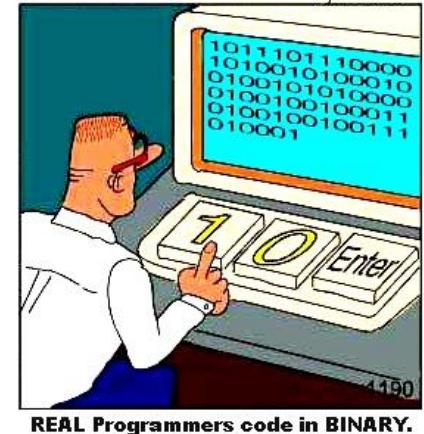
# Capacity of Secondary Storage Devices

- **Kilobyte (KB or K)** – about 1 thousand bytes
- **Megabyte (MB or M or Meg)** – about 1 million bytes
- **Gigabyte (GB or Gig)** – about 1 billion bytes
- **Terabyte (TB)** – about 1 trillion bytes



# Software

- Programs – instructions that tell the computer what to do
- Categories
  - **Application software** - enables you to solve specific problems or perform specific tasks.
  - **System software** - handles tasks specific to technology management and coordinates the interaction of all technology devices
  - **Utility software** - provides additional functionality to your operating system software

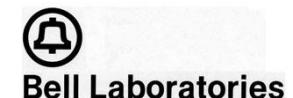


# System Software

- Operating System
  - UNIX / Linux
  - Windows
  - MAC OS
  - Palm OS
  - Android
- Language Translators
  - C, C++, Basic, Java, ...
- Device Drivers



# C Programming Language

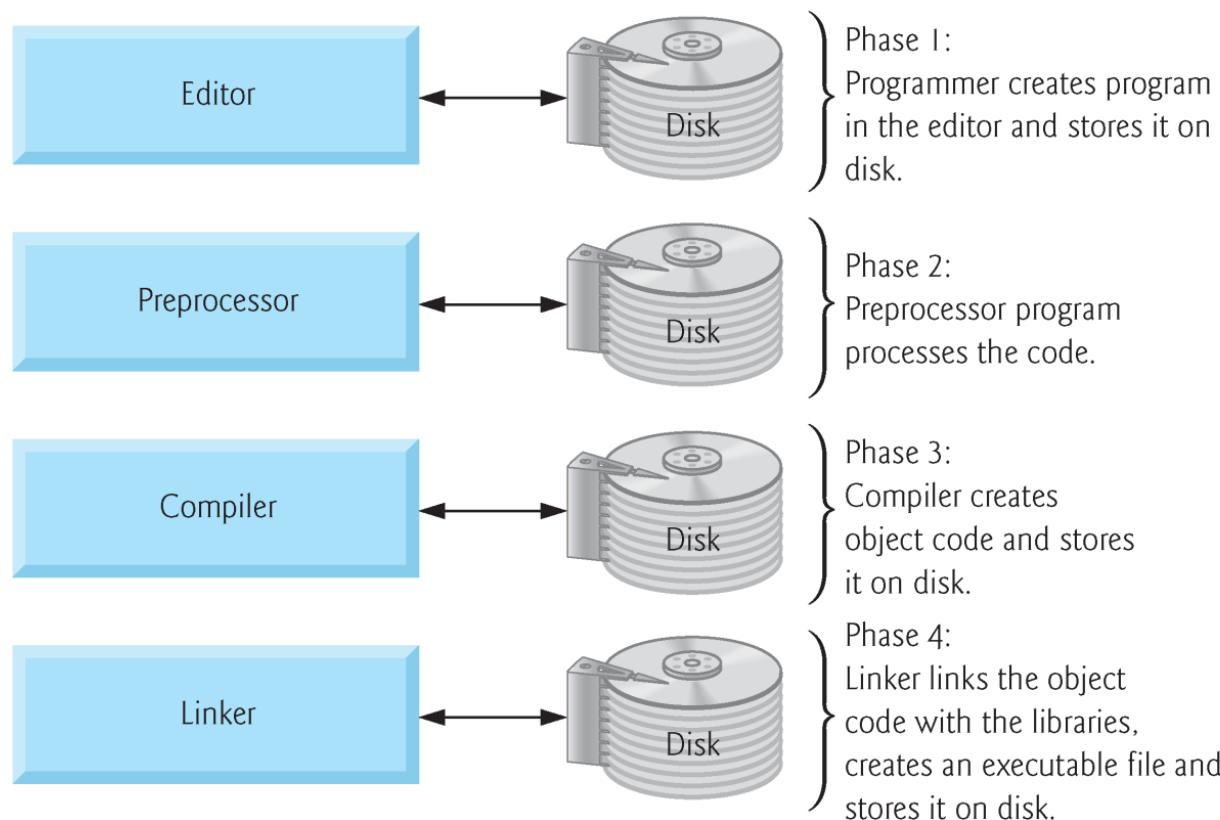


- Developed at AT&T Bell Labs in early 1970s
- Unix also developed at Bell Labs
  - All but core of Unix is in C
- Standardized by  
American National Standards Institute (ANSI)

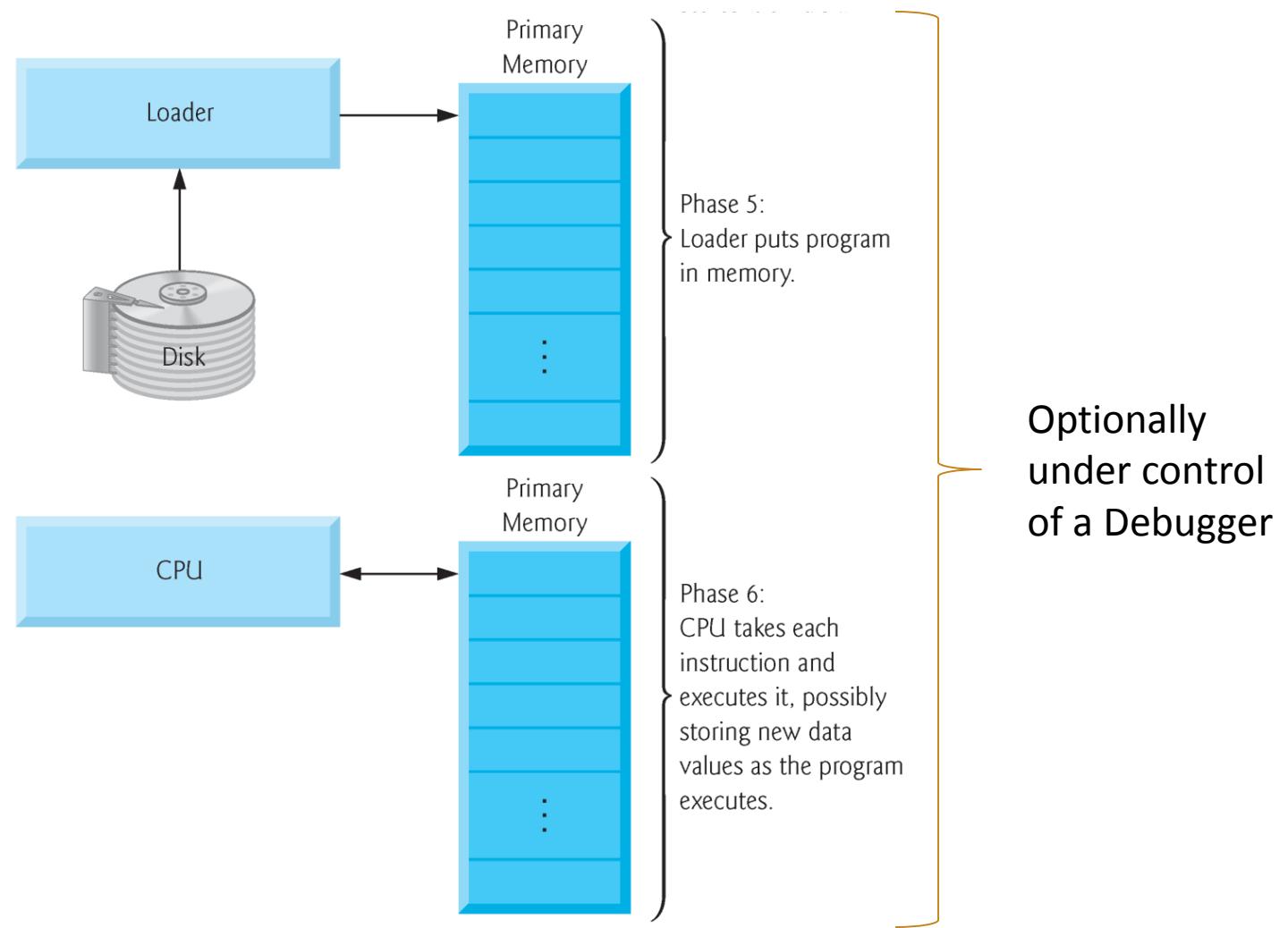


*Because C is a hardware-independent, widely available language, applications written in C can run with little or no modifications on a wide range of different computer systems.*

# C Development Environment

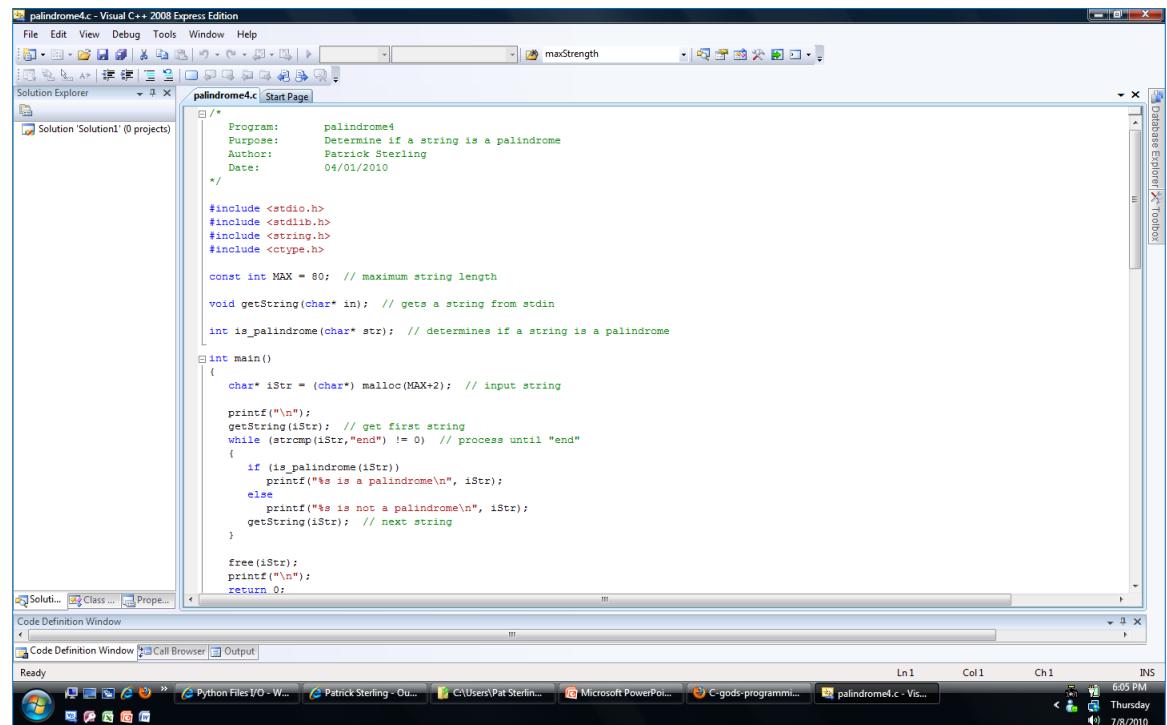


# Execution Environment



# IDE

- Integrated Development Environment
  - Editor
  - Compiler
  - Debugger
- Ex:
  - MS Visual C++
  - Xcode



The screenshot shows the Microsoft Visual Studio 2008 Express Edition interface. The main window displays a C++ code editor with the file name "palindrome4.c". The code is a program to determine if a string is a palindrome. It includes comments at the top providing purpose, author, and date. The code uses standard C headers and defines a maximum string length of 80. It reads strings from standard input, processes them until it finds a string ending with "end", and then prints whether the string is a palindrome or not. The code editor has syntax highlighting for C keywords and operators.

```
/*  
 * Program: palindrome4  
 * Purpose: Determine if a string is a palindrome  
 * Author: Patrick Sterling  
 * Date: 04/01/2010  
 */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <ctype.h>  
  
const int MAX = 80; // maximum string length  
  
void getString(char* in); // gets a string from stdin  
  
int is_palindrome(char* str); // determines if a string is a palindrome  
  
int main()  
{  
    char* iStr = (char*) malloc(MAX+2); // input string  
  
    printf("\n");  
    getString(iStr); // get first string  
    while (strcmp(iStr, "end") != 0) // process until "end"  
    {  
        if (is_palindrome(iStr))  
            printf("%s is a palindrome\n", iStr);  
        else  
            printf("%s is not a palindrome\n", iStr);  
        getString(iStr); // next string  
    }  
  
    free(iStr);  
    printf("\n");  
    return 0;  
}
```

# Best Programming Language?



God's Programming Language

```
#include <stdio.h>
#define IN 1 /* inside a word */
#define OUT 0 /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}

#include <stdio.h>
#define IN 1 /* inside a word */
#define OUT 0 /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}

#include <stdio.h>
#define IN 1 /* inside a word */
#define OUT 0 /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

# Programming in C



## Chapter 1 Introduction to C

*THE END*

# Programming in C



## *Chapter 2 Your First Program*

Hello World!

Soon I will  
control the world!



# Introduction to C



- C language
  - Facilitates a structured and disciplined approach to computer program design
  - Provides low-level access
  - Highly portable

# Program Basics

- The ***source code*** for a program is the set of instructions written in a high-level, human readable language.

**X = 0;**

**MOVE 0 TO X.**

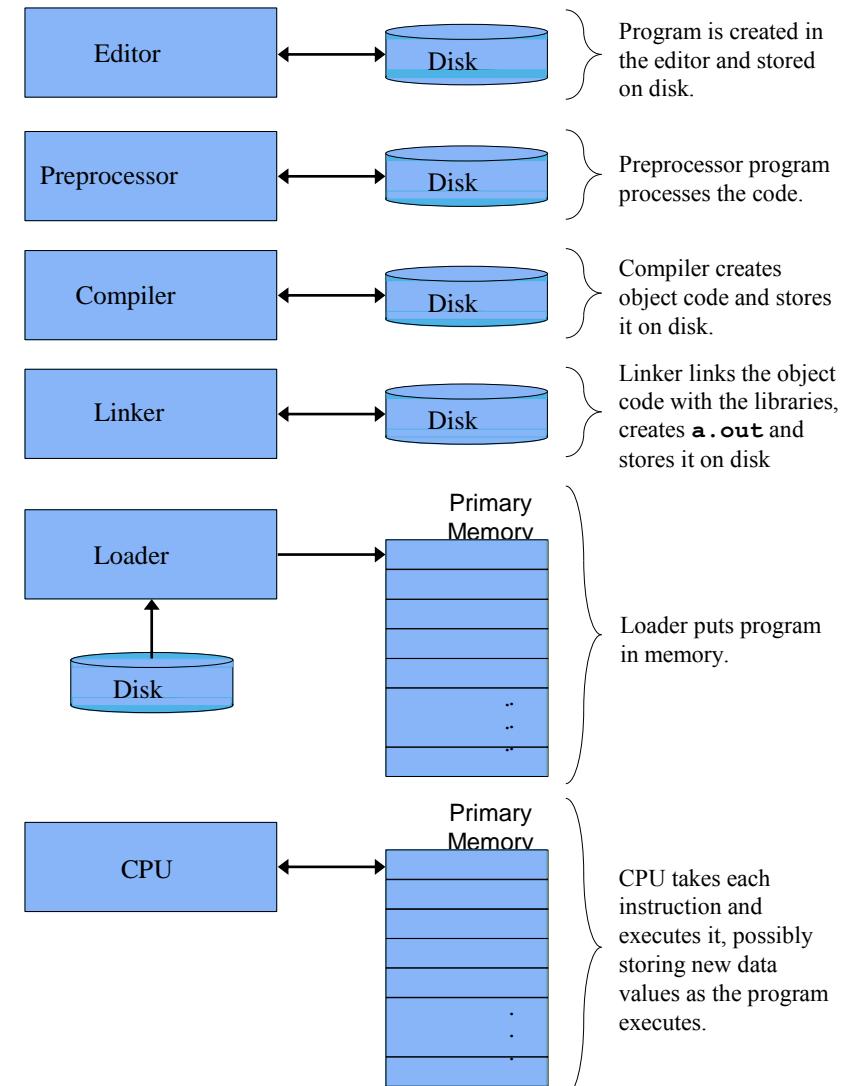
**X := 0**

- The source code is transformed into ***object code*** by a ***compiler***. Object code is a machine usable format.
- The computer ***executes*** a program in response to a command.

# Basics of a Typical C Environment

## Phases of C Programs:

1. Edit
2. Preprocess
3. Compile
4. Link
5. Load
6. Execute



# GCC Program Basics

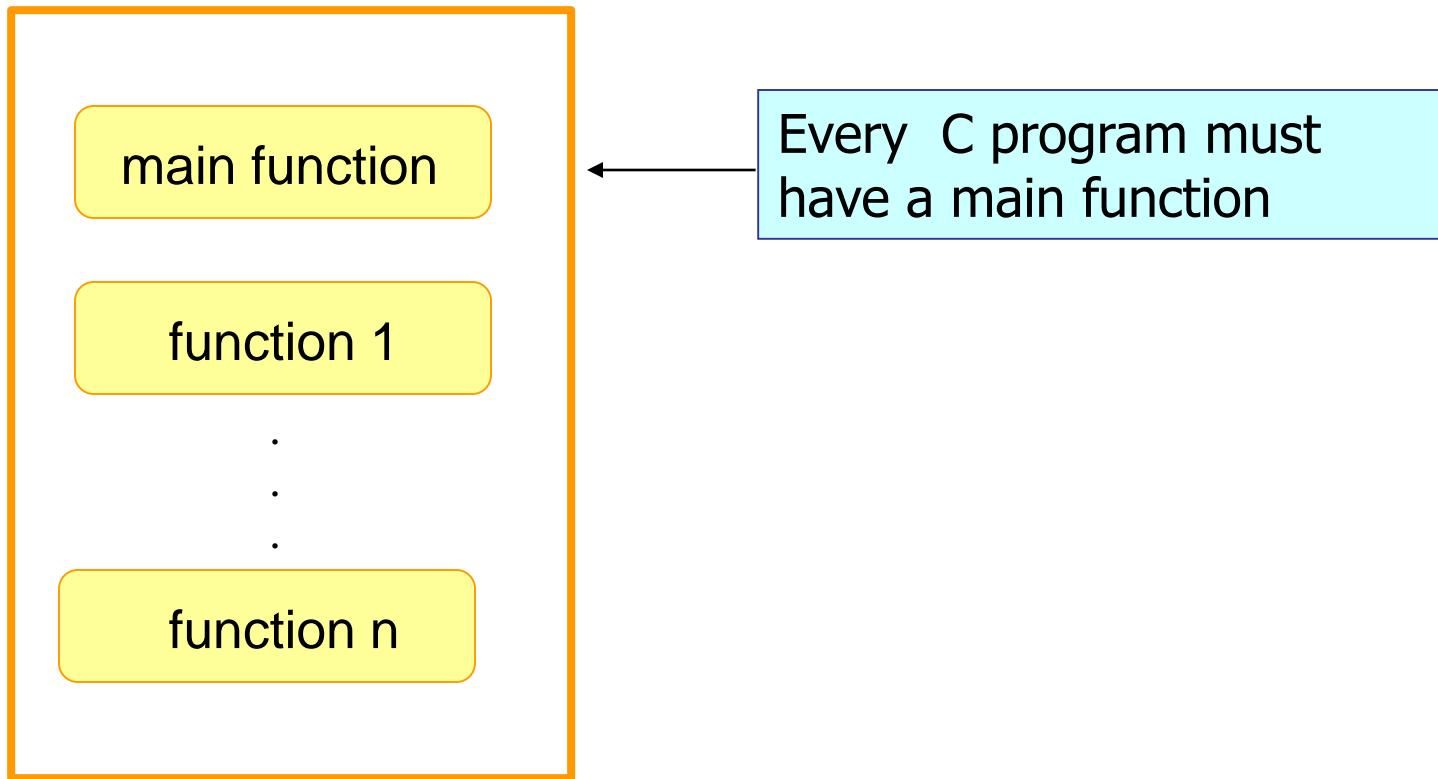
- The basic program writing sequence:
  1. create or modify a file of instructions using an editor
    - Unix: Pico, vi, gEdit, emacs, ...
  2. compile the instructions with GCC
  3. execute or run the compiled program
  - repeat the sequence if there are mistakes



Pico:

<http://www.bgsu.edu/departments/compsci/docs/pico.html>

# Structure of a C Program



# Functions

- Each function consists of a **header** followed by a **basic block**.
- General format:

**<return-type> fn-name (parameter-list)**        
*basic block*

# The Basic Block

```
{  
    declaration of variables  
    executable statements  
}
```

- A semi-colon (;) is used to terminate a statement
- A block consists of zero or more statements
- Nesting of blocks is legal and common
  - Each interior block may include variable declarations

# Return statement

- **return *expression***
  1. Sets the return value to the value of the expression
  2. Returns to the caller / invoker
- Example:

```
int main()          // header
{
    .
    return 0;       // program ending successfully
}
```

# SSH Secure Shell

- On-Campus / VPN
  - SSH to one of the machines in the list
  - machine.cs.clemson.edu
- Off-Campus
  - SSH to access.cs.clemson.edu
  - ssh machine.cs.clemson.edu

```
This copy of SSH Secure Shell is a non-commercial version.  
This version does not include PKI and PKCS #11 functionality.  
  
Last login: Mon Jul 12 17:35:28 2010 from 71-12-9-167.dhc  
*****  
Welcome to the  
CLEMSON UNIVERSITY SCHOOL OF COMPUTING  
Unauthorized use is prohibited!  
  
PARTIAL LIST OF PUBLIC CLIENT MACHINES  
Hostnames OS/Arch  
-----  
dragon1 - dragon24 CentOS 5/x86_64  
frog1 - frog27 CentOS 5/x86_64  
gecko1 - gecko22 CentOS 5/x86_64  
sparc1 - sparc2 Solaris 10/SPARC  
  
* Questions or problems regarding Unix systems should be addressed to  
"ithelp@clemson.edu" or the friendly folks in 109 or 137 McAdams  
  
***** Last updated: 2010-07-06 *****  
[17:36:47] userid@sparc1:~ [99]  
Connected to access.cs.clemson.edu SSH2 - aes128-cbc - hmac-md5 - nc 80x24
```

# Unix Commands: mkdir & cd

**mkdir cpsc1110**

- Creates a new directory / folder

**cd cpsc1110**

- Changes the current directory

**pico ch02First.c**

- Runs the pico editor to edit file ch02First.c

# Our First Program

```
// Program:      ch03First
// Purpose:      A first program in C
//               Printing a line of text
// Author:       Ima Programmer
// Date:        mm/dd/yy

#include <stdio.h>

int main() {
    printf("Go Tigers!!!\n");
    return 0; //indicates program ended successfully
}
```

Go Tigers!!!

# Compiling and Running a Program

- To compile and print all warning messages, type

*gcc -Wall prog-name.c*

- If using math library (math.h), type

*gcc -Wall prog-name.c -lm*  After

- By default, the compiler produces the file *a.out*

# Compiling and Running a Program

- To execute the program type

`./a.out`

- The `./` indicates the current directory

- To specify the file for the object code, for example, `p1.o`, type

`gcc -Wall prog1.c -o p1.o`

then type

`./p1.o`

to execute the program

# Comments

- Make programs easy to read and modify
- Ignored by the C compiler
- Two methods:
  1. // - line comment
    - everything on the line following // is ignored

```
//Purpose:    Display Go Tigers!
```

2. /\* \*/ - block comment
  - everything between /\* \*/ is ignored

```
/*
Program:    ch02First
Purpose:    Display Go Tigers!
Author:    Ima Programmer
Date:    mm/dd/yy
*/
```

# Preprocessor Directive: #include

- A C program line beginning with # that is processed by the compiler before translation begins.
- #include pulls another file into the source
  - `#include <stdio.h>` causes the contents of the named file, stdio.h, to be inserted where the # appears. File is commonly called a header file.
    - <>'s indicate that it is a compiler standard header file.
  - `#include "myfunctions.h"` causes the contents of myfunctions.h to be inserted
    - “”s indicate that it is a user file from current or specified directory

**#include: Chapter 12 p. 311**

# Introduction to Input/Output

- *Input data* is read into variables
- *Output data* is written from variables.
- Initially, we will assume that the user
  - enters data via the terminal keyboard
  - views output data in a terminal window on the screen

```
[16:35:02] psterli@frog6:~/cpsc111 [104] ./a.out
Enter two integers: 6 20
Enter a floating point number: 3.5
6 / 20 = 0
3.50 / 20 = 0.17
sqrt(3.500000) = 1.87
[16:35:35] psterli@frog6:~/cpsc111 [105]
```

# Program Input / Output

- The C run-time system automatically opens two files for you at the time your program starts:
  - **stdin** – standard input (from the keyboard)
  - **stdout** – standard output (to the terminal window in which the program started)
- Later, how to read and write files on disk
  1. Using stdin and stdout
  2. Using FILE's

# Console Input/Output

- Defined in the C library included in <stdio.h>
  - Must have this line near start of file:  
**#include <stdio.h>**
  - Includes input functions scanf, fscanf, ...
  - Includes output functions printf, fprintf, ...

# Console Output - printf

- Print to standard output,  
typically the screen
- General format (value-list may not be required):  
`printf("format string", value-list);`

```
printf("Go Tigers!!!!");
```

# Console Output

What can be output?

- Any data can be output to display screen
  - Literal values
  - Variables
  - Constants
  - Expressions (which can include all of above)
- Note
  - Values are passed to printf
  - Addresses are passed to scanf

# Console Output

- We can
  - Control vertical spacing with blank lines
    - Use the escape sequence "\n", new line
      - Should use at the end of all lines unless you are building lines with multiple printf's.
      - If you printf without a \n and the program crashes, you will not see the output.
  - Control horizontal spacing
    - Spaces
    - Use the escape sequence "\t", tab
      - Sometimes undependable.

# Terminal Output - Examples

```
printf("Hello World!\n");
```

- Sends string "Hello World" to display, skipping to next line

```
printf("Good morning\nMs Smith.\n");
```

- Displays the lines  
    Good morning  
    Ms Smith.

# Program Output: Escape Character \

- Indicates that a “special” character is to be output

Escape Sequence	Description
\n	Newline. Position the screen cursor to the beginning of the next line.
\t	Horizontal tab. Move the screen cursor to the next tab stop.
\r	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert. Sound the system bell.
\\\	Backslash. Used to print a backslash character.
\"	Double quote. Used to print a double quote character.

# Template: a.c

- Starting point for a new program
  - Read into (^R in pico) or
  - Copy into (cp command) a new file
    - Ex: cp a.c prog1.c

```
/*
Program:    ?
Purpose:   ?
Author:    Im A Programmer
Date:     mm/dd/yy
*/
#include <stdio.h>

int main() {
    return 0; // Return normally
}
```

# Programming in C



## *Chapter 2* *Your First Program*

*THE END*

# Programming in C



## Chapter 3

### Variables and Expressions

A dark blue, textured background resembling a chalkboard. Two mathematical equations are written in white chalk:
$$a^2 + b^2 = c^2$$
$$m^2 + n^2 = c^2$$
Both equations have a small red checkmark in the top right corner.

# Reserved Words and Identifiers

- Reserved word
  - Word that has a specific meaning in C
    - Ex: int, return
- Identifier
  - Word used to name and refer to a data element or object manipulated by the program.



# Valid Identifier Names

- Begins with a letter or underscore symbol
- Consists of letters, digits, or underscores only
- Cannot be a C reserved word
- Case sensitive
  - Total ≠ total ≠ TOTAL
- Examples:

distance  
milesPerHour  
\_voltage  
goodChoice  
high\_level  
MIN\_RATE

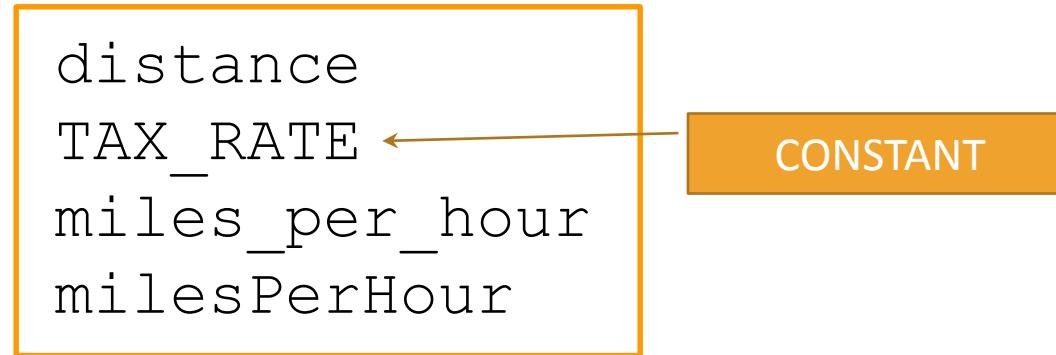
# Invalid Identifier Names

- Does not begin with a letter or underscore symbol or
- Contains other than letters, digits, and underscore or
- Is a C reserved word
- Examples

```
x-ray  
2ndGrade  
$amount  
two&four  
after five  
return
```

# Identifier Name Conventions

- Standard practice, not required by C language
  - Normally lower case
  - Constants upper case
- Multi-word
  - Underscore between words or
  - Camel case - each word after first is capitalized



# Variable



- Name is a valid identifier name
- Is a memory location where a value can be stored for use by a program
- Value can change during program execution
- Can hold only one value
  - Whenever a new value is placed into a variable, the new value replaces the previous value.



# Variables Names

- C: Must be a valid identifier name
- C: Variables must be declared with a name and a data type *before* they can be used in a program
- Should not be the name of a standard function or variable
- Should be descriptive; the name should be reflective of the variable's use in the program
  - For class, make that must be descriptive except subscripts
- Abbreviations should be commonly understood
  - Ex. amt = amount

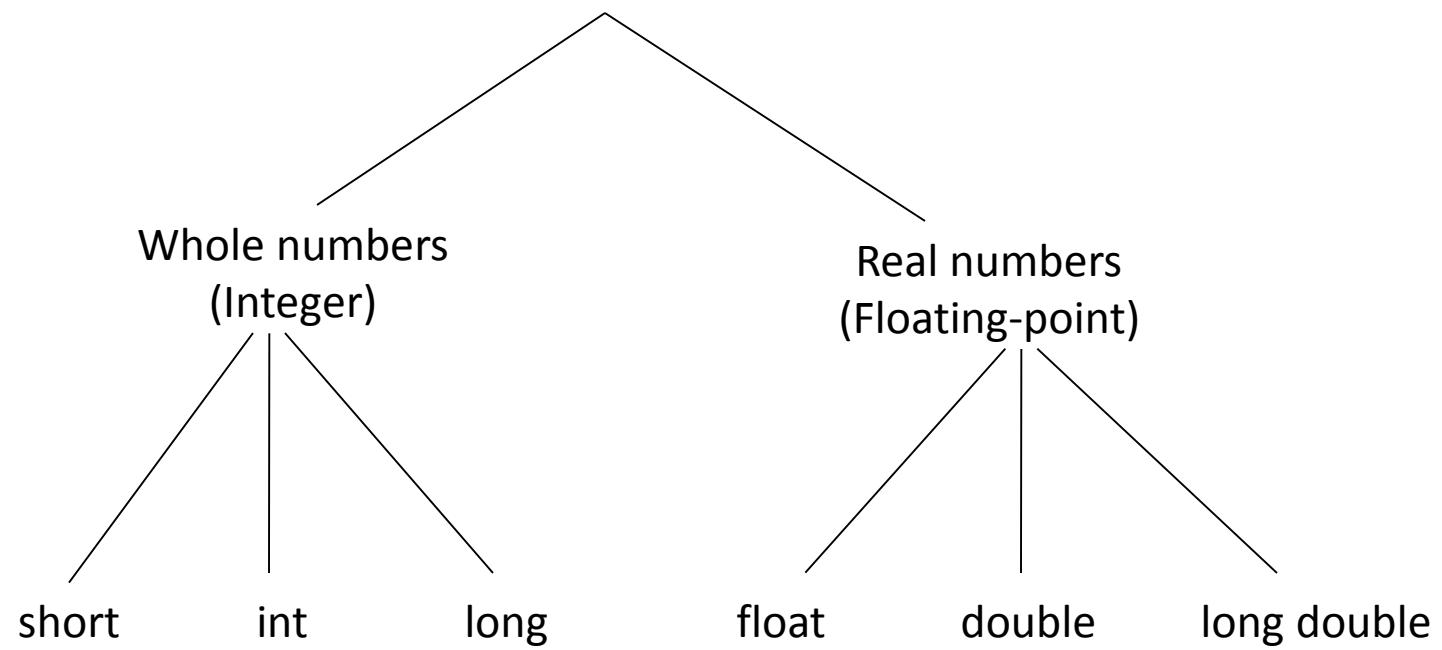
# Variable/Named Constant Declaration Syntax

```
optional_modifier  data_type  name_list;
```

- *optional\_modifier* – type modifier
  - Used to distinguish between **signed** and **unsigned** integers
    - The default is signed
  - Used to specify size (**short**, **long**)
  - Used to specify named constant with **const** keyword
- *data\_type* - specifies the type of value; allows the compiler to know what operations are valid and how to represent a particular value in memory
- *name\_list* – program identifier names
- Examples:

```
int test-score;
const float TAX_RATE = 6.5;
```

# Numeric Data Types



# Data Types and Typical Sizes

Type Name	Memory Used	Size Range	Precision	Guarantee
short (= short int)	2 bytes	-32,768 to 32,767	N/A	16 bits
int	4 bytes	-2,147,483,648 to 2,147,483,647	N/A	16 bits
long (= long int)	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	N/A	32 bits
float	4 bytes	approximately $10^{-38}$ to $10^{38}$	7 digits	6 digits
double	8 bytes	approximately $10^{-308}$ to $10^{308}$	15 digits	10 digits
long double	10 bytes	approximately $10^{-4932}$ to $10^{4932}$	19 digits	10 digits

# Determining Data Type Size

- `sizeof` operator
  - Returns size of operand in bytes
  - Operand can be a data type
- Examples:

```
sizeof(int)  
sizeof(double)
```



# Characters

Type Name	Memory Used	Sample Size Range
char	1 byte	All ASCII characters

ASCII = American Standard Code for Information Interchange

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

# Boolean Data Type

- Data type: `_Bool`
  - Can only store 0 & 1
  - Non zero value will be stored as 1
- Data type : `bool`
  - `<stdbool.h>` defines `bool`, `true`, and `false`
- Any expression
  - 0 is false
  - Non-zero is true



Basic Data Types: Table 4.1 p. 30

More types: Table A.4 p. 431

# Variable Declaration Examples

```
int age;  
  
short first_reading;  
short int last_reading;  
  
long first_ssn;  
long int last_ssn;  
  
float interest_rate;  
double division_sales;  
  
char grade, midInitial;
```

# Assigning Values to Variables

- Allocated variables without initialization have an undefined value.
- We will use three methods for assigning a value to a variable
  - Initial value
    - In the declaration statement
  - Processing
    - the assignment statement
  - Input
    - scanf function

# Initializing Variables

- Initializing variables in declaration statements

```
int age = 22;
double rate = 0.75;
char vowel = 'a';
int count = 0, total = 0;
```

# Assignment Operator =



- Assigns a value to a variable
- Binary operator (has two operands)
- Not the same as "equal to" in mathematics
- General Form:

**l\_value = r\_value**

- Most common examples of l\_values (left-side)
  - A simple variable
  - A pointer dereference (in later chapters)
- r\_values (right side) can be any valid expression
- Assignment expression has value of assignment
  - Allows us to do something like

**a = b = 0;**

# Example Assignment Statement

- Statement

```
x = y + 5;
```

5 is literal value  
or constant

- Means:  
Evaluate the expression on the right and put the result in the memory location named x
- If the value stored in y is 18,  
then 23 will be stored in x

# Other Example Assignments

- Example:

```
distance = rate * time;
```

l\_value: distance

r\_value: rate \* time

- Other Examples:

```
pay = 65.75;  
hourly_rate = pay / hours;
```



# Terminal Output

What can be output?

- Any data can be output to standard output (stdout), the terminal display screen
  - Literal values
  - Variables
  - Constants
  - Expressions (which can include all of above)
- printf function:  
*The values of the variables are passed to printf*

# Syntax: printf function

```
printf(format_string, expression_list)
```

- Format\_string specifies how expressions are to be printed
  - Contains placeholders for each expression
    - Placeholders begin with % and end with type
- Expression list is a list of zero or more expressions separated by commas
- Returns number of characters printed

# Typical Integer Placeholders

- %d or %i - for integers, %l for long

```
printf("%d", age);  
printf("%l", big_num);
```

- %o - for integers in octal

```
printf("%o", a);
```

- %x – for integers in hexadecimal

```
printf("%x", b);
```

# Floating-point Placeholders

- %f, %e, %g – for float
  - %f – displays value in a standard manner.
  - %e – displays value in scientific notation.
  - %g – causes printf to choose between %f and %e and to automatically remove trailing zeroes.
- %lf – for double (the letter l, not the number 1)

# Printing the value of a variable

- We can also include literal values that will appear in the output.
  - Use two %'s to print a single percent

\n is new line

```
printf("x = %d\n", x);
printf("%d + %d = %d\n", x, y, x+y);
printf("Rate is %d%%\n", rate*100);
```

# Output Formatting Placeholder

`%[flags][width][.precision][length]type`

- Flags
  - left-justify
  - + generate a plus sign for positive values
  - # puts a leading 0 on an octal value and 0x on a hex value
  - 0 pad a number with leading zeros
- Width
  - Minimum number of characters to generate
- Precision
  - Float: Round to specified decimal places

# Output Formatting Placeholder

`%[flags] [width] [.precision] [length] type`

- Length
  - I long
- Type
  - d, i decimal unsigned int
  - f float
  - x hexadecimal
  - o octal
  - % print a %

# Output Formatting Placeholder

`%[flags] [width] [.precision] [length] type`

- Examples:

```
printf("[%5d] [%+05d] [%#5o] [%#7x]\n",
       123, 123, 123, 123);
printf("[%f] [%5.2f] [%5.0f%%]\n",
       123.456, 123.456, 123.456);
```

```
[ 123] [+0123] [ 0173] [    0x7b]
[123.456000] [123.46] [ 123%]
```

Format codes w/printf:

<http://en.wikipedia.org/wiki/Printf>

# Return from printf

- A successful completion of printf returns the number of characters printed. Consequently, for the following:

```
int num1 = 55;
int num2 = 30;
int sum = num1 + num2;
int printCount;
printCount = printf("%d + %d = %d\n", num1, num2, sum);
```

if printf() is successful,  
the value in printCount should be 13.

# Literals / Literal Constants

- Literal – a name for a specific value
- Literals are often called constants
- Literals do not change value

*literal*

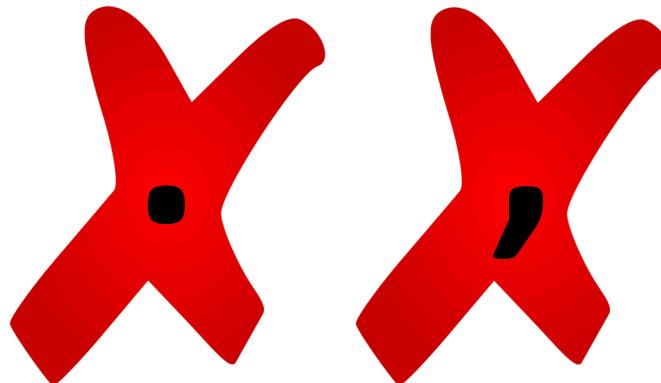
# Integer Constants

- Must not contain a decimal point
- Must not contain a comma
- Examples

-25

68

17895



# Integer Constants

- May be expressed in several ways

decimal number

120

hexadecimal number

0x78

octal number

0170

ASCII encoded character 'x'

119 77 167 «#119; w  
120 78 170 «#120; x  
121 79 171 «#121; Y

- All of the above represent the 8-bit byte whose value is 01111000

# Integer Constants

- Constants of different representations may be intermixed in expressions:
  - Examples

```
x = 5 + 'a' - 011 + '\n';  
x = 0x51 + 0xc + 0x3d + 0x8;
```

# Floating Point Constants

- Contain a decimal point.
- Must not contain a comma
- Can be expressed in two ways

decimal number: 23.8 4.0

scientific notation: 1.25E10



# char Constants

- Enclosed in apostrophes, single quotes
- Examples:

'a'

'A'

'\$'

'2'

'+'

- Format specification: %c

# String Constants

- Enclosed in quotes, double quotes
- Examples:

"Hello"

"The rain in Spain"

"x"

- Format specification/placeholder: %s

# Terminal Input

- We can put data into variables from the standard input device (`stdin`), the terminal keyboard
- When the computer gets data from the terminal, the user is said to be acting interactively.
- Putting data into variables from the standard input device is accomplished via the use of the `scanf` function



# Keyboard Input using scanf

- General format

`scanf(format-string, address-list)`

- Example

```
scanf("%d", &age);
```

& (address of operator)  
is required

& & \$&

- The format string contains placeholders (one per address) to be used in converting the input.
  - %d – Tells *scanf* that the program is expecting an ASCII encoded integer number to be typed in, and that *scanf* should convert the string of ASCII characters to internal binary integer representation.
- Address-list: List of memory addresses to hold the input values

# Addresses in scanf()

```
scanf ("%d", &age);
```

- Address-list must consist of addresses only
  - scanf() puts the value read into the memory address
  - The variable, age, is not an address; it refers to the *content* of the memory that was assigned to age
- & (address of) operator causes the **address of the variable** to be passed to *scanf* rather than the value in the variable
- Format string should consist of a placeholder for each address in the address-list

Format codes w/scanf:

<http://en.wikipedia.org/wiki/Scanf>

# Return from scanf()

- A successful completion of scanf() returns the number of input values read. Returns EOF if hits end-of-file reading one item.

Consequently, we could have

```
int dataCount;  
dataCount = scanf ("%d %d", &height, &weight);
```

- If scanf() is successful,  
the value in dataCount should be 2
- Spaces or new lines separate one value from another

# Keyboard Input using scanf

- When using scanf for the terminal, it is best to first issue a prompt

```
printf("Enter the person's age: ");
scanf("%d", &age);
```

- Waits for user input, then stores the input value in the memory space that was assigned to number.
- Note: '\n' was omitted in printf
  - Prompt 'waits' on same line for keyboard input.
- Including printf prompt before scanf maximizes user-friendly input/output

# scanf Example

```
int main() {
    // declare variables
    int x;
    int y;
    int sum;

    // read values for x and y from standard input
    printf("Enter value for x: ");
    scanf("%d", &x);

    printf("Enter value for y: ");
    scanf("%d", &y);

    sum = x + y;

    // print
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    printf("x + y = %d\n", sum);

    printf("%d + %d = %d\n", x, y, sum);
    printf("%d - %d = %d\n", x, y, (x - y));
    printf("%d * %d = %d\n", x, y, (x * y));
    return 0;
}
```

# Input using scanf()

- Instead of using scanf() twice, we can use one scanf() to read both values.

```
int main() {
    // declare variables
    int x;
    int y;
    int sum;

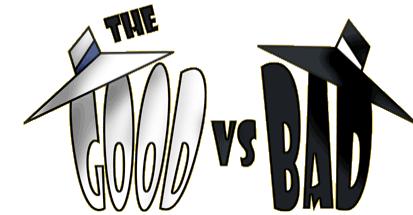
    // read values for x and y from standard input
    printf("\n");
    printf("Enter values for x and y: ");
    scanf("%d %d", &x, &y);

    sum = x + y;

    // print
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    printf("x + y = %d\n", sum);

    printf("%d + %d = %d\n", x, y, sum);
    printf("%d - %d = %d\n", x, y, (x - y));
    printf("%d * %d = %d\n", x, y, (x * y));

    printf("\n");
    return 0;
}
```



# Bad Data

```
[11:34:55] psterli@access:~/cpsc111 [112] gcc ch04Scan2.c -Wall
[11:34:57] psterli@access:~/cpsc111 [113] ./a.out
Enter values for x and y: 24 m6
x = 24
y = 4
x + y = 28
24 + 4 = 28
24 - 4 = 20
24 * 4 = 96
[11:35:24] psterli@access:~/cpsc111 [114]
```

- `scanf` stops at the first bad character.
- The value of `y` was never set. The value 4 is what was left in the memory location named `num2` the last time the location was assigned a value.

# Format Placeholder for Input

- When reading data, use the following format specifiers / placeholders
  - %d - for integers, no octal or hexadecimal
  - %i – for integers allowing octal and hexadecimal
  - %f - for float
  - %lf – for double (the letter l, not the number 1)
- Do not specify width and other special printf options

# Executable Code

- Expressions consist of legal combinations of
  - constants
  - variables
  - operators
  - function calls



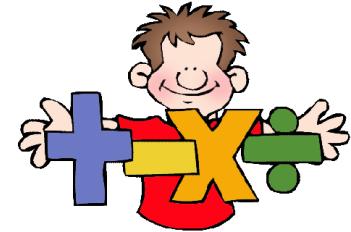
# Executable Code

- Operators

- Arithmetic: +, -, \*, /, %
- Relational: ==, !=, <, <=, >, >=
- Logical: !, &&, ||
- Bitwise: &, |, ~, ^
- Shift: <<, >>

- See Expressions

- 4<sup>th</sup> Edition: p. 443-450
- 3<sup>rd</sup> Edition: p. 439-445



# Arithmetic

- Rules of operator precedence (arithmetic ops):

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they are evaluated left to right.
* , /, or %	Multiplication Division Modulus	Evaluated second. If there are several, they are evaluated left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

- Average  $a + b + c / 3$  ?

# Precedence Example

- Find the average of three variables a, b and c

Do not use:  $a + b + c / 3$

Use:  $(a + b + c) / 3$

# The Division Operator

- Generates a result that is the same data type of the largest operand used in the operation.
- Dividing two integers yields an integer result.  
Fractional part is truncated.

$$5 / 2 \rightarrow 2$$

$$17 / 5 \rightarrow 3$$

Handwritten division diagram:

$$\begin{array}{r} 193 \\ 5 ) 965 \\ -5 \\ \hline 46 \\ -45 \\ \hline 15 \end{array}$$

15 ÷ 5 = 3

➤ Watch out: You will not be warned!

# The Division Operator

- Dividing one or more decimal floating-point values yields a decimal result.

$5.0 / 2 \rightarrow 2.5$

$4.0 / 2.0 \rightarrow 2.0$

$17.0 / 5.0 \rightarrow 3.4$

# The modulus operator: %

- % modulus operator returns the remainder

$$7 \% 5 \rightarrow 2$$

$$5 \% 7 \rightarrow 5$$

$$12 \% 3 \rightarrow 0$$

$$\begin{array}{r} 193 \\ 5 ) 965 \\ -5 \\ \hline 46 \\ -45 \\ \hline 15 \end{array}$$

15 ÷ 5 = 3

# Evaluating Arithmetic Expressions

- Calculations are done ‘one-by-one’ using precedence, left to right within same precedence
  - $11 / 2 / 2.0 / 2$  performs 3 separate divisions.
    1.  $11 / 2 \rightarrow 5$
    2.  $5 / 2.0 \rightarrow 2.5$
    3.  $2.5 / 2 \rightarrow 1.25$



# Arithmetic Expressions

math expression

$$\frac{a}{b}$$

$$2x$$

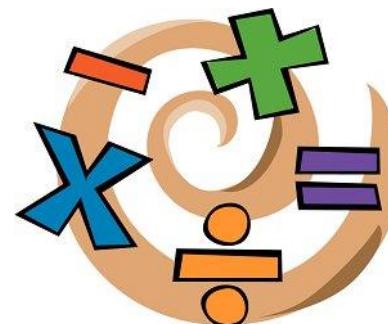
$$\frac{x - 7}{2 + 3y}$$

C expression

$$a/b$$

$$2*x$$

$$(x-7)/(2 + 3*y)$$



# Evaluating Arithmetic Expressions

$$2 * (-3) \quad -6$$

$$4 * 5 - 15 \quad 5$$

$$4 + 2 * 5 \quad 14$$

$$7 / 2 \quad 3$$

$$7 / 2.0 \quad 3.5$$

$$2 / 5 \quad 0$$

$$2.0 / 5.0 \quad 0.4$$

$$2 / 5 * 5 \quad 0$$

$$2.0 + 1.0 + 5 / 2 \quad 5.0$$

$$5 \% 2 \quad 1$$

$$4 * 5/2 + 5 \% 2 \quad 11$$

# Data Assignment Rules

- In C, when a floating-point value is assigned to an integer variable, the decimal portion is truncated.

```
int grams;  
grams = 2.99;      // 2 is assigned to variable grams!
```

- Only integer part ‘fits’, so that’s all that goes
- Called ‘implicit’ or ‘automatic type conversion’



# Arithmetic Precision

- Precision of Calculations
  - VERY important consideration!
    - Expressions in C might not evaluate as you ‘expect’!
    - ‘Highest-order operand’ determines type of arithmetic ‘precision’ performed
    - Common pitfall!
    - Must examine each operation

change

# Type Casting

- Casting for Variables
  - Can add '.0' to literals to force precision arithmetic, but what about variables?
    - We can't use '`myInt . 0`'!
- type cast – a way of changing a value of one type to a value of another type.
- Consider the expression `1/2`: In C this expression evaluates to 0 because both operands are of type integer.

# Type Casting

1 / 2.0 gives a result of 0.5

Given the following:

```
int m = 1;  
int n = 2;  
int result = m / n;
```

result is 0, because of integer division

# Type Casting

- To get floating point-division, you must do a type cast from int to double (or another floating-point type), such as the following:

```
int m = 1;  
int n = 2;  
double doubleAnswer = (double) m / n;
```

Type cast operator

- This is different from `(double) (m/n)`

# Type Casting

- Two types of casting
  - Implicit – also called ‘Automatic’
    - Done for you, automatically  
`17 / 5.5`

This expression causes an ‘implicit type cast’ to take place, casting the 17 → 17.0
  - Explicit type conversion
    - Programmer specifies conversion with cast operator  
`(double)17 / 5.5`  
`(double) myInt / myDouble`

# Abreviated/Shortcut Assignment Operators

- Shortcut

- Assignment expression abbreviations

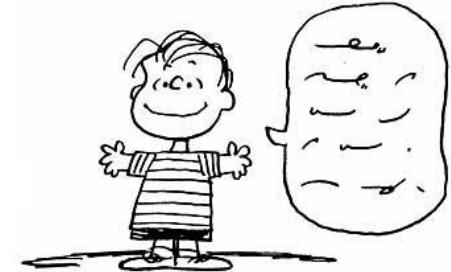
`a = a + 3;` can be abbreviated as `a += 3;`  
using the addition assignment operator



- Examples of other assignment operators include:

Assignment	Shortcut
<code>d = d - 4</code>	<code>d -= 4</code>
<code>e = e * 5</code>	<code>e *= 5</code>
<code>f = f / 3</code>	<code>f /= 3</code>
<code>g = g % 9</code>	<code>g %= 9</code>

# Shorthand Operators



- Increment & Decrement Operators
  - Just short-hand notation
  - Increment operator, ++  
`intVar++;` is equivalent to  
`intVar = intVar + 1;`
  - Decrement operator, --  
`intVar--;` is equivalent to  
`intVar = intVar - 1;`

# Shorthand Operators: Two Options

- Post-Increment  
**x++**
  - Uses current value of variable,  
THEN increments it
- Pre-Increment  
**++x**
  - Increments variable first,  
THEN uses new value

**POST**

**PRE**

# Shorthand Operators: Two Options

- ‘Use’ is defined as whatever ‘context’ variable is currently in
- No difference if ‘alone’ in statement:  
`x++;` and `++x;` → identical result

# Post-Increment in Action

POST  
INCREMENT

- Post-Increment in Expressions:

```
int n = 2;
int result;
result = 2 * (n++);
printf("%d\n", result);
printf("%d\n", n);
```

- This code segment produces the output:  
4  
3
- Since post-increment was used

# Pre-Increment in Action

PRE  
IN  
C

- Now using pre-increment:

```
int n = 2;  
int result;  
result = 2 * (++n);  
printf("%d\n", result);  
printf("%d\n", n);
```

- This code segment produces the output:  
6  
3
- Because pre-increment was used

# Programming in C



## Chapter 3

### Variables and Expressions

*THE END*

# Programming in C



## Chapter 4A

### Repetition/Looping



**Repetition**  
**Repetition**  
**Repetition**  
**Repetition**

**Repetition**  
**Repetition**  
**Repetition**  
**Repetition**  
**Repetition** ...



# Example 1

```
// Read two integers and print sum  
int num1, num2, sum;  
  
scanf("%d %d", &num1, &num2);  
sum = num1 + num2;  
printf("%d + %d = %d\n", num1, num2, sum);
```

- What if we want to process three different pairs of integers?



# Example 2

- One solution is to copy and paste the necessary lines of code. Consider the following modification:

```
scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);

scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);

scanf("%d %d", &num1, &num2);
sum = num1 + num2;
printf("%d + %d = %d\n", num1, num2, sum);
```

- What if you wanted to process four sets?  
Five? Six? ....



# Processing an arbitrary number of pairs

- We might be willing to copy and paste to process a small number of pairs of integers but
- How about 1,000,000 pairs of integers?
- The solution lies in mechanisms used to **control the flow of execution**
- In particular, the solution lies in the constructs that allow us to instruct the computer to **perform a task repetitively**



# Repetition (Looping)

- Use looping when you want to execute a block of code several times
  - Block of code = Body of loop
- C provides three types of loops



*while* statement

- *Most flexible*
- *No 'restrictions'*



*for* statement

- *Natural 'counting' loop*



*do-while* statement

- *Always executes body at least once*

# The while Repetition Structure

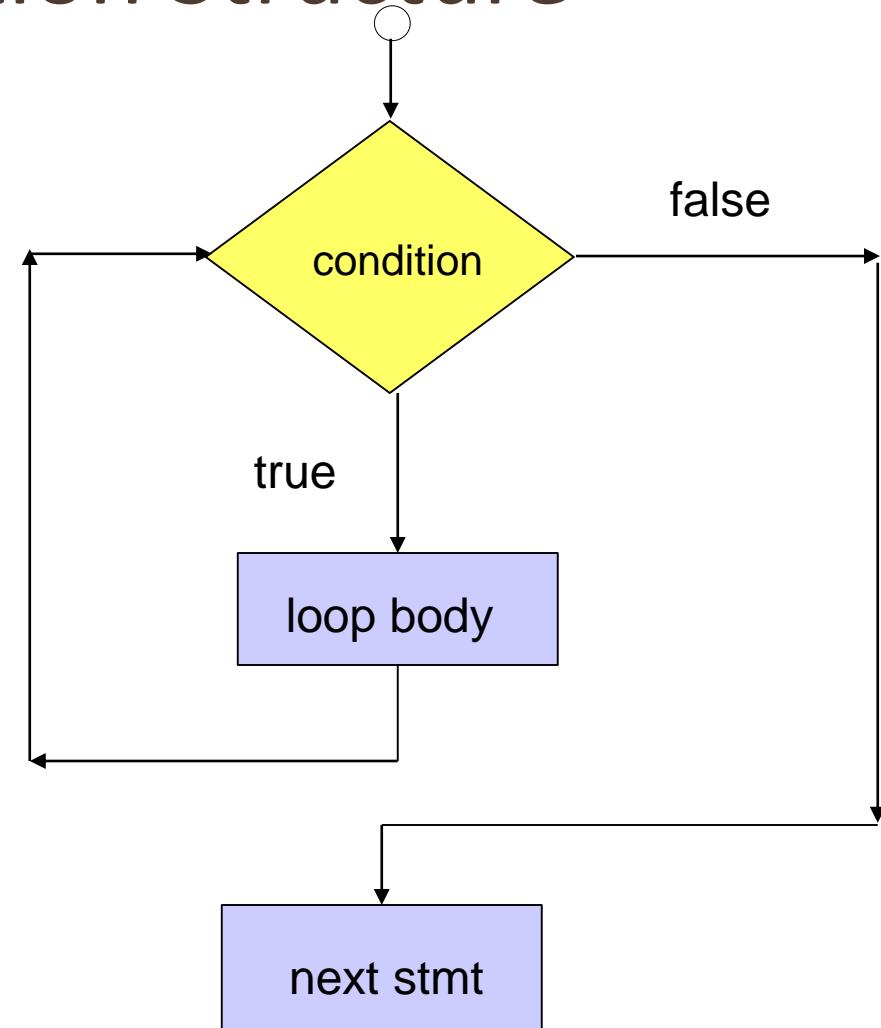
- Repetition structure
  - Programmer specifies
    - Condition under which actions will be executed
    - Actions to be repeated
  - Psuedocode

*While there are more items on my shopping list  
Purchase next item and cross it off my list*
- **while** loop repeated
  - As long as condition is true
  - Until condition becomes false



# The while Repetition Structure

- The condition is tested
- If the condition is true, the loop body is executed and the condition is retested.
- When the condition is false, the loop is exited.





# The while Repetition Structure

- Syntax:

```
while (expression)
    basic block
```

- Expression = Condition to be tested
  - Resolves to true or false
- Basic Block = Loop Body
  - Reminder – Basic Block:
    - Single statement or
    - Multiple statements enclosed in braces

# Loop Control Variable (LCV)

- The loop control variable is the variable whose value controls loop repetition.
- For a while loop to execute properly, the loop control variable must be
  - declared
  - initialized
  - tested
  - updated in the body of the loop in such a way that the expression/condition will become false
    - If not we will have an endless or infinite loop

# Counter-Controlled Repetition

- Requires:
  1. Counter variable , LCV, initialized to beginning value
  2. Condition that tests for the final value of the counter (i.e., whether looping should continue)
  3. Constant increment (or decrement) by which the control variable is modified each time through the loop
- Definite repetition
  - Loop executes a specified number of times
  - Number of repetitions is known

# Example 3

```
int count;                                // LCV: Loop Control Variable
int num1, num2, sum;

count = 0;                                 // 1. Initialize LCV
while (count < 5) {                         // 2. Test LCV
    scanf("%d %d", &num1, &num2);
    sum = num1 + num2;
    printf("%d + %d = %d\n", num1, num2, sum);
    count = count + 1;                      // 3. Increment LCV
}
```

EXECUTION CHART

count	count<5	repetition
0	true	1
1	true	2
2	true	3
3	true	4
4	true	5
5	false	

# Loop Pitfalls

```
// Echo numbers entered back to user
printf("Enter number or zero to end: ");
scanf("%d", &num);
while (num != 0);
{
    printf("Number is %d\n\n", num);
    printf("Enter another number or zero to end: ");
    scanf("%d", &num);
}
```

**Enter value or zero to end: 2**



**What is wrong with my  
program? It just sits there!**

# Loop Pitfalls: Misplaced semicolon

```
// Echo numbers entered back to user
printf("Enter number or zero to end: ");
scanf("%d", &num);
while (num != 0);           ←
{
    printf("Number is %d\n\n", num);
    printf("Enter another number or zero to end: ");
    scanf("%d", &num);
}
```

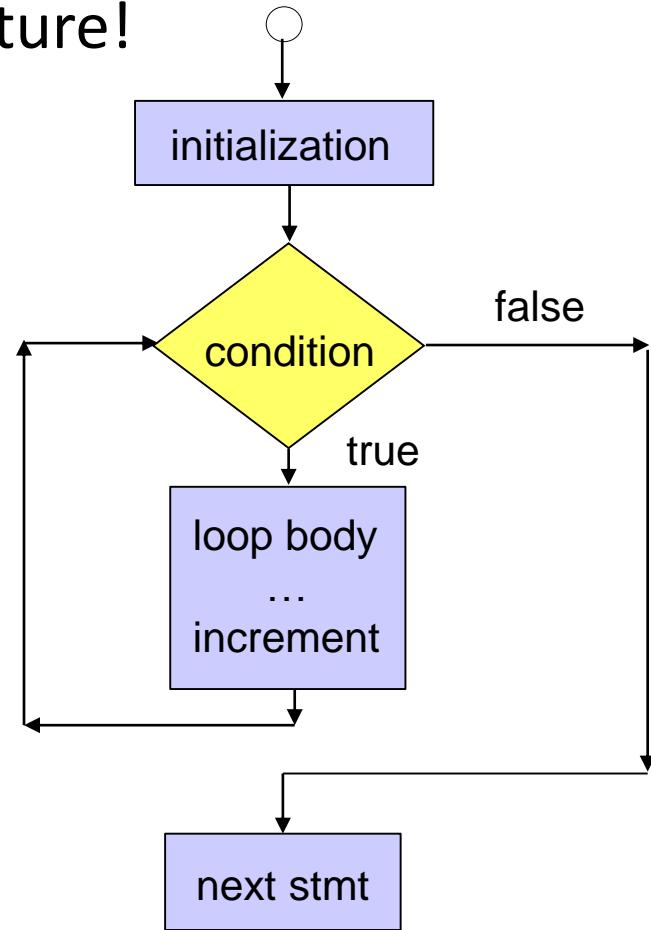
Do not place semi-colon here!

- Notice the ';' after the while condition!
  - Body of loop is between ) and ;
- Result here: **INFINITE LOOP!**  
**Ctrl-c = Kill foreground process**



# The for Repetition Structure

- A natural 'counting' loop
- Steps are built into for structure!
  1. Initialization
  2. Loop condition test
  3. Increment or decrement



# Review: Assignment Operators

- Statements of the form

variable = variable *operator* expression;

can be rewritten as

variable *operator*= expression;

- Examples of assignment operators:

a += 5              (a = a + 5)

a -= 4              (a = a - 4)

b \*= 5              (b = b \* 5)

c /= 3              (c = c / 3)

d %= 9              (d = d % 9)

# Review: Pre-increment operator

Pre-increment operator: `++n`

i) Stand alone: add 1 to n

If n equals 1, then after execution of the statement

```
++n;
```

the value of n will be 2.

ii) In an expression:

Add 1 to n and then use the new value of n in the expression.

```
printf ("%d", ++n);
```

If n is initially 1, the above statement will print the value 2.

After execution of printf, n will have the value 2.

# Review: Post-increment operator

Pre-increment operator: n++

i) Stand alone: add 1 to n

If n equals 1, then after execution of the statement

```
n++;
```

the value of n will be 2.

ii) In an expression:

Use the value of n in the expression and then add 1 to n.

```
printf ("%d", n++);
```

If n is initially 1, the above statement will print the value 1 and then add 1 to n. After execution, n will have the value 2.

# Pre- and Post-decrement operator

- Pre- and post-decrement operators, `--n`, `n--` , behave in a similar manner
- **Use caution when using in an expression**
  - Do not use unless you know what you are doing!



# The *for* Repetition Structure

- Syntax:

```
for (initialization; test; increment)  
    basic block
```

# *for* loop example

- Prints the integers from one to ten

```
int counter;
for (counter = 1; counter <= 10; counter++)
{
    printf("%d\n", counter);
}
```

```
int counter;
counter = 1;
while (counter <= 10)
{
    printf("%d\n", counter);
    counter++;
}
```

# for Loop Example

How many times does loop body execute?

```
int count;
for (count = 0; count < 3; count++) {
    printf("Bite %d -- ", count+1);
    printf("Yum!\n");
}
```



# for Loop Example

How many times does loop body execute?

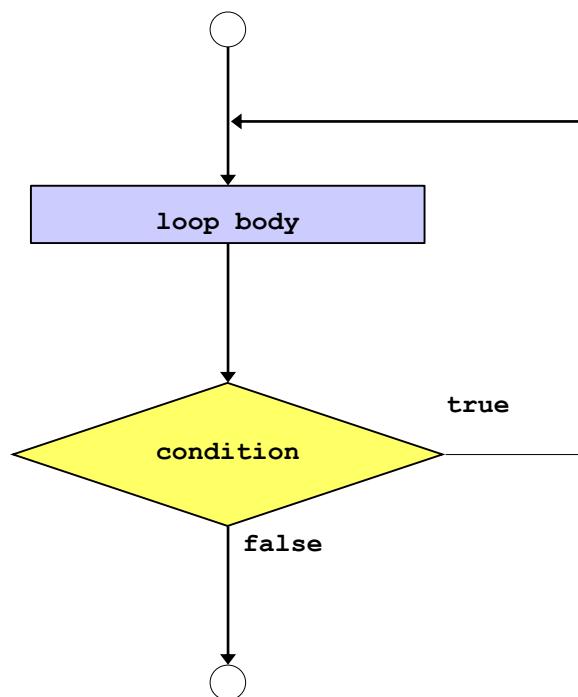
```
int count;
for (count = 0; count < 3; count++) {
    printf("Bite %d -- ", count+1);
    printf("Yum!\n");
}
```

Bite 1 -- Yum!  
Bite 2 -- Yum!  
Bite 3 -- Yum!



# The do-while Repetition Structure

- The **do-while** repetition structure is similar to the **while** structure
  - Condition for repetition tested after the body of the loop is executed



*All actions are performed at least once!*



# The do-while Repetition Structure

- Syntax:

```
do {  
    statements  
} while ( condition );
```

# The do-while Repetition Structure

- Example

```
int counter = 1;  
do {  
    printf("%d\n", counter);  
    counter++;  
} while (counter <= 10);
```

- Prints the integers from 1 to 10

# The do-while Repetition Structure

- Example

```
do {  
    printf("Enter a positive weight: ");  
    scanf("%d", &weight);  
} while (weight <= 0);
```

- Makes sure that the user enters a valid weight



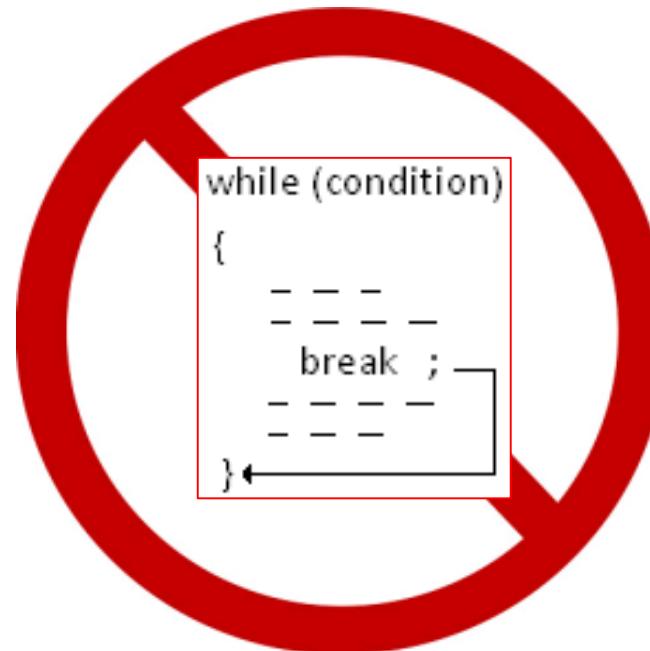
# The break Statement

- **break**

- Causes immediate exit from a **while**, **for**, **do/while** or **switch** structure
- **We will use the break statement only to exit the switch structure!**



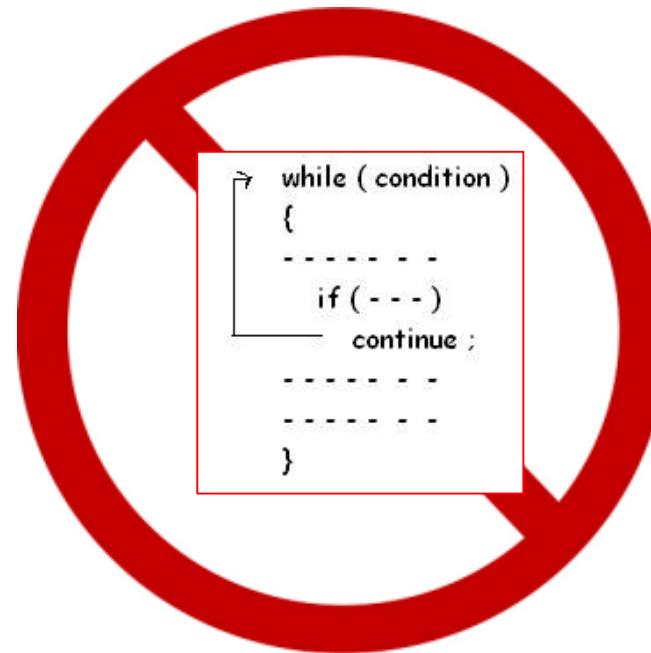
<http://mmp0109.c60145.e62>



# The continue Statement

- **continue**

- Control passes to the next iteration
- **We will not use the continue statement!**



# Programming in C



## Chapter 4A Repetition/Looping

*THE END*

# Programming in C



## Chapter 4B

### Looping Subtasks



# Looping Subtasks

- We will examine some basic algorithms that use the while and if constructs. These subtasks include
  - Reading unknown quantity of data
  - Counting things
  - Accumulating (summing) totals
  - Searching for specific values
  - Finding extreme values

# Looping Subtasks

- Examples will be based upon common models:

Priming Read

or

Input Count

*Initialize program state*

*Read the first value (priming read)*

*While (data exists)*

*update program state as needed*

*read next value(s)*

*Output final state*

*Initialize program state*

*While (input count OK)*

*update program state as needed*

*Output final state*

- The type of state that must be maintained by the program depends on the nature of the problem and can include:
  - *indicator (true/false) variables*
  - *counter variables*
  - *sum variables*
  - *previous input value variables*

# Counter-Controlled Repetition

- Number of items is known before loop

```
// Read and print 5 test scores
int count, score;
for (count = 1; count <= 5; count++) {
    scanf("%d", &score);
    printf("Score %d is %d\n", count, score);
}
```

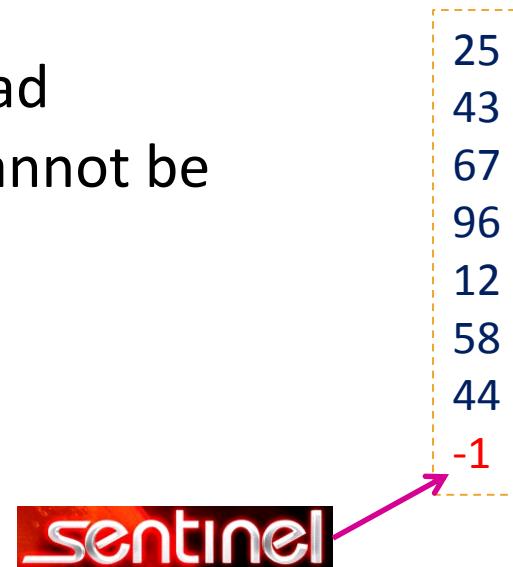
- Suppose the problem becomes:

*Develop a class-averaging program that will process an arbitrary number of grade scores each time the program is run.*



# Sentinel-Controlled Repetition

- One way to handle an arbitrary number of input values is to have the user enter a special value to indicate the end of input.
- Such a value is a sentinel value.
  - Indicates end of valid input
  - Loop ends when sentinel value is read
  - Must choose a sentinel value that cannot be confused with a regular input value.

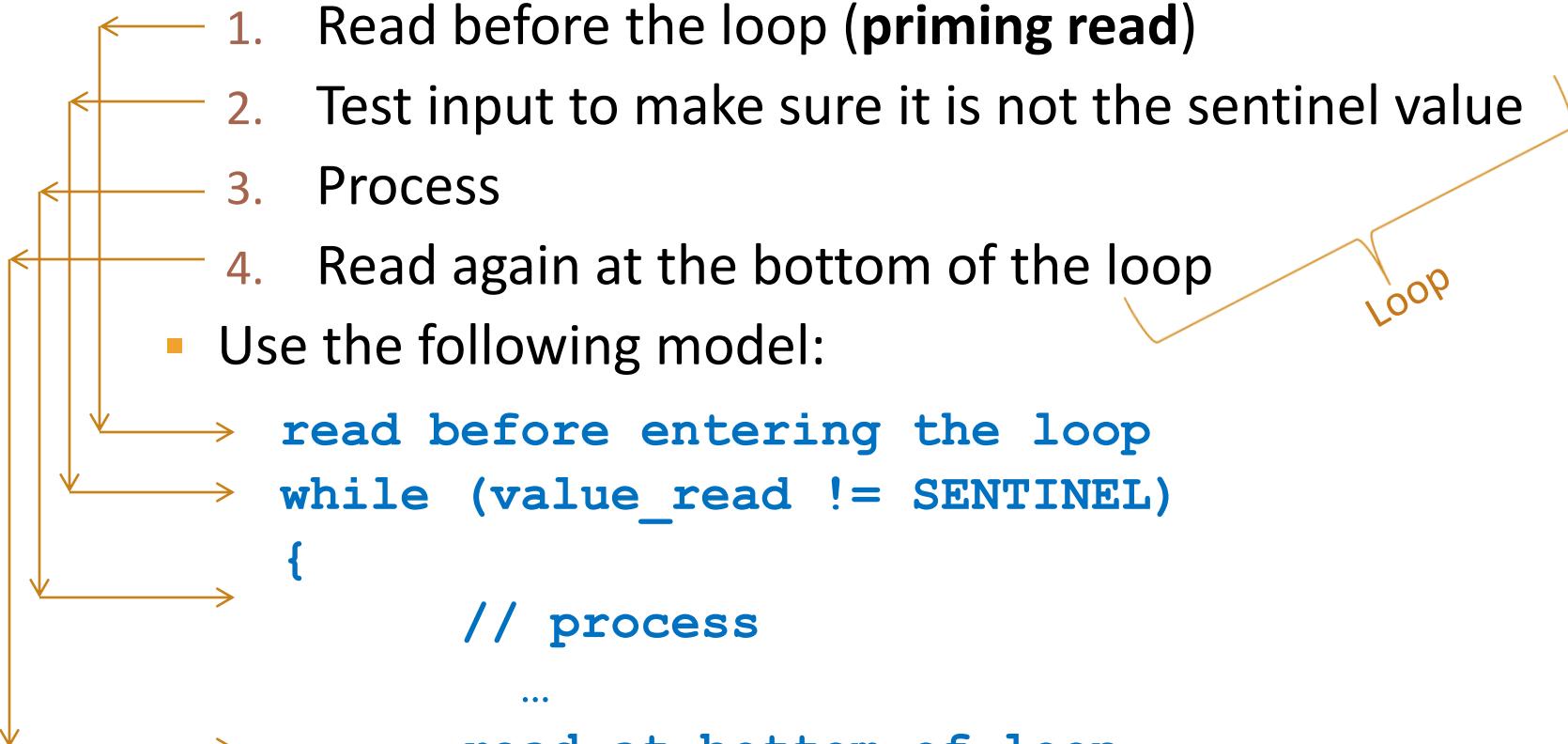


# Sentinel-Controlled Priming Read

- For sentinel-controlled loops

1. Read before the loop (**priming read**)
  2. Test input to make sure it is not the sentinel value
  3. Process
  4. Read again at the bottom of the loop
- Use the following model:

```
read before entering the loop
while (value_read != SENTINEL)
{
    // process
    ...
    read at bottom of loop
        (before entering loop again)
}
```



# Sentinel-Controlled Loop using Priming Read

25  
43  
67  
96  
12  
58  
44  
**-1**

**sentinel**

```
// Read and print numbers using priming read
int num;

scanf("%d", &num);           // Priming read
while (num != -1) {           // Sentinel is -1
    printf("%d\n", num);
    scanf("%d", &num);        // Read another number
}
```

# Sentinel-Controlled Loop using Input Count

25  
43  
67  
96  
12  
58  
44  
**-1**

**sentinel**

```
// Read and print numbers using input count
int inputCount; // Items read
int num;

while (scanf("%d", &num) == 1 && num != -1) { // Sentinel is -1
    printf("%d\n", num);
}
```

# Example of sentinel-controlled loop

```
25 43  
67 96  
12 58  
44 99  
-1
```

**sentinel**

```
// Read pairs and print sums
int num1, num2, sum;

scanf("%d", &num1);           // Priming read
while (num1 != -1) {          // Sentinel is -1
    scanf("%d", &num2);       // Read second number
    sum = num1 + num2;
    printf("%d + %d = %d\n", num1, num2, sum);
    scanf("%d", &num1);       // Read another first number
}
```

# Processing an arbitrary number of pairs

- Sometimes it is not possible to find a sentinel value
- We can use
  - End-of-input controlled loops
    - Uses return from scanf
    - Can be fooled by invalid data
  - End-of-file controlled loops
    - Uses function feof



# End of Data

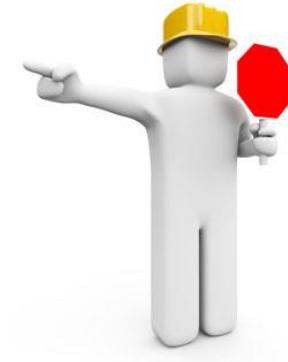
- Hardware & Software  
End-Of-File
  - Keyboard
    - Ctrl-d (Does not work on Mac!)

25 43  
67 96  
12 58  
44 99  
Ctrl-d



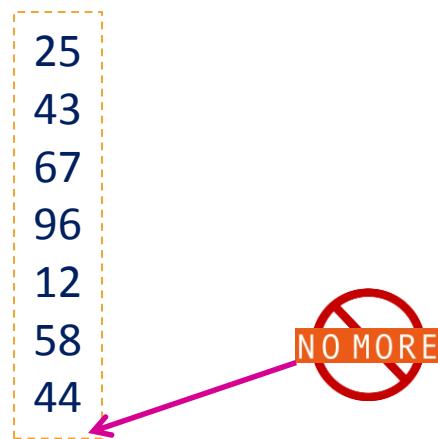
The End Is Here!

# Redirection



- Redirection: Read / Write to actual file
  - stdin: **cmd < input-file**
    - Ex: ./a.out < nums.txt
  - stdout: **cmd > output-file**
    - Ex: ./a.out > report.txt
  - stdout (append): **cmd >> output-file**
    - Ex: ./a.out >> report.txt
  - Both: **cmd < input-file > output-file**
    - Ex: ./a.out < nums.txt > report.txt
  - Leave out prompts when designing for redirection

# Example: End-of-input controlled loop using items read & priming read



```
// Read and print using items read & priming read
int inputCount; // Items read
int num;

inputCount = scanf("%d", &num); // Priming read
while (inputCount == 1) { // Check count
    printf("%d\n", num);
    inputCount = scanf("%d", &num); // Read another number
}
```

# Example: End-of-input controlled loop using just items read

25  
43  
67  
96  
12  
58  
44

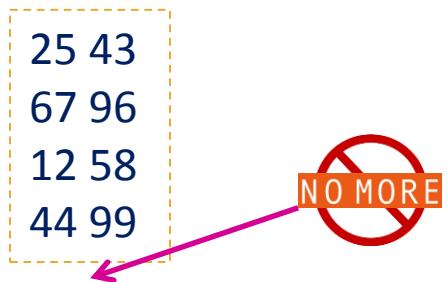
```
// Read and print using just items read
int num;

while (scanf("%d", &num) == 1) { // Check count
    printf("%d\n", num);
}

//or
while (scanf("%d", &num) != EOF) { // Check for EOF
    printf("%d\n", num);
}
```



# Example: End-of-input controlled loop using number of items read



```
// Read pairs and print sums using items read
int num1, num2, sum;

while (scanf("%d %d", &num1, &num2) == 2) {      // Check items read
    sum = num1 + num2;
    printf("%d + %d = %d\n", num1, num2, sum);
}
```

# Detecting End-of-File

- Function: `feof`
  - Syntax: **feof(file-pointer)**
    - Returns true or false
    - Standard input: `feof(stdin)`
  - Use in a while loop -  
`while (!feof(stdin))`

# Example: End-of-file controlled loop

```
25  
43  
67  
96  
12  
58  
44
```

End of File

```
// Read and print numbers using EOF  
int num;  
  
scanf("%d", &num);           // Priming read  
while (!feof(stdin)) {      // Check for EOF  
    printf("%d\n", num);  
    scanf("%d", &num);        // Read another number  
}
```

# Example: end-of-file controlled loop

```
25 43  
67 96  
12 58  
44 99
```

End of File

```
// Read pairs and print sums using end-of-file  
int num1, num2, sum;  
  
scanf("%d", &num1);           // Priming read  
while (!feof(stdin)) {       // Check for EOF  
    scanf("%d", &num2);       // Read second number  
    sum = num1 + num2;  
    printf("%d + %d = %d\n", num1, num2, sum);  
    scanf("%d", &num1);       // Read another first number  
}
```

# Looping Subtask: Counting

- Example: Find the number of scores in a file
  - Here the program state that must be maintained is a *counter* that maintains the number of scores that have been read so far.
- Steps
  - Declare an int variable for the count
  - Initialize the count to zero
  - Increment the count in the body of the loop



# Looping Subtask: Counting

```
// Print score count w/priming read
int scoreCount;           // counter
int score;

scoreCount = 0;           // initialize counter

printf("Enter first score or ctrl-d to end: ");
scanf("%d", &score);
while (!feof(stdin)) {
    scoreCount++;        // increment counter
    scanf("%d", &score);
    printf("Enter next score or ctrl-d to end: ");
}

printf("Score count is %d\n", scoreCount);
```

# Looping Subtask: Counting

```
// Print score count w/scanf in while
int scoreCount;           // counter
int score;

scoreCount = 0;           // initialize counter

printf("Enter first score or ctrl-d to end: ");
while (scanf("%d", &score) == 1) {
    scoreCount++;        // increment counter
    printf("Enter next score or ctrl-d to end: ");
}

printf("Score count is %d\n", scoreCount);
```

# Looping Subtask: Counting

```
// Print score count w/for
int scoreCount;           // counter
int score;

scoreCount = 0;           // initialize counter

printf("Enter first score or ctrl-d to end: ");
for (scoreCount = 0; scanf("%d", &score) == 1, scoreCount++)
    printf("Enter next score or ctrl-d to end: ");

printf("Score count is %d\n", scoreCount);
```

# Looping Subtask: Counting

```
// Print score count w/for & no prompts
int scoreCount;           // counter
int score;

scoreCount = 0;           // initialize counter

for (scoreCount = 0; scanf("%d", &score) == 1, scoreCount++)
    /*null*/ ;

printf("Score count is %d\n", scoreCount);
```

# Counting Example

- What if we want to print the number of passing scores (scores  $\geq 70$ )?
  - We need a mechanism that allows us to count only if the score is greater than or equal to 70
  - Use *if* stmt

# Looping Subtask: Counting

```
// Print passing score count
int passCount;           // passing counter
int score;

passCount = 0;           // initialize counter

scanf("%d", &score);
while (!feof(stdin)) {
    if (score >= 70)
        passCount++;      // increment pass counter
    scanf("%d", &score);
}

printf("Passing score count is %d\n", passCount);
```

# Counting Example

- What if we want to print the number of passing scores (scores  $\geq 70$ ) and the number of failing scores?
  - Use *if -else*

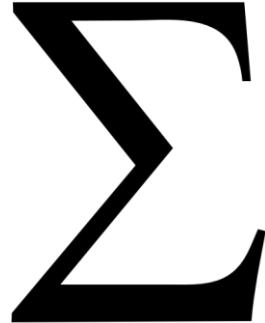
# Looping Subtask: Counting

```
// Print passing and failing score count
int passCount;           // passing counter
int failCount;           // failing counter
int score;

passCount = 0;            // initialize counters
failCount = 0;

scanf("%d", &score);
while (!feof(stdin)) {
    if (score >= 70)
        passCount++;      // increment pass counter
    else
        failCount++;      // increment fail counter
    scanf("%d", &score);
}

printf("Passing score count is %d\n", passCount);
printf("Failing score count is %d\n", failCount);
```



# Looping Subtask: Accumulation (Summing)

- The state that must be maintained is the sum of all values that have been seen so far.
  - Declare a variable to hold the sum (accumulator)
  - Initialize the sum to zero
  - In the body of the loop, add the new value to the sum

# Accumulating Example

```
// Print score sum
int scoreSum;           // total accumulator
int score;

scoreSum = 0;           // initialize total

scanf("%d", &score);
while (!feof(stdin)) {
    scoreSum += score; // add score to total
    scanf("%d", &score);
}

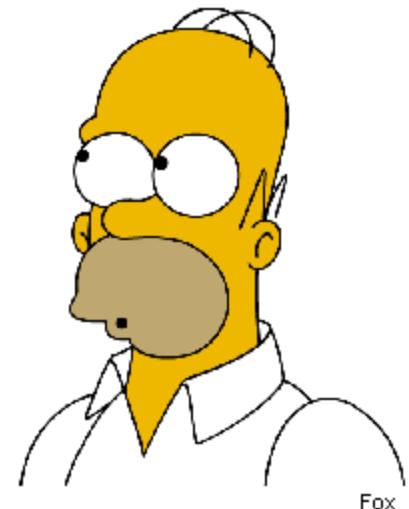
printf("Score total is %d\n", scoreSum);
```

# Counting & Accumulating Example

- Problem

- Problem
  - *A class of ten students took a quiz.*
  - *The grades (integers in the range 0 to 100) for this quiz are available to you.*
  - *Determine the class average on the quiz.*

- Hint: Requirements for an average
  - Count of number of items
  - Sum of the items



Fox

# Counting & Accumulating Example

- Pseudocode:

*Set total to zero*

*Set grade counter to one*

*While grade counter is less than or equal to ten*

*Input the next grade*

*Add the grade into the total*

*Add one to the grade counter*

*Set the class average to the total divided by ten*

*Print the class average*

**EXCELLENT**



# Looping Subtasks: Searching



- Need a variable to indicate whether or not the program has encountered the target value, call it *found*
- Initialize *found* to 0 (false)
- Each time through the loop, check to see if the current value equals the target value
  - If so, assign 1 to *found*

# Searching Exercise

Write a C program that

1. Reads a target score at the beginning of the file
2. Reads a set of scores and determines if the target score is in the set of scores
3. If found prints

**Target ## was found**

otherwise prints

**Target ## was not found**

# Looping Subtasks: Searching

```
// Determine if target score is found
int score, target;
int found = 0; // found = false

scanf("%d", &target);

scanf("%d", &score);
while (!feof(stdin)) {
    if (score == target)
        found = 1; // found = true
    scanf("%d", &score);
}

if (found)
    printf("Target %d was found\n", target);
else
    printf("Target %d was not found\n", target);
```

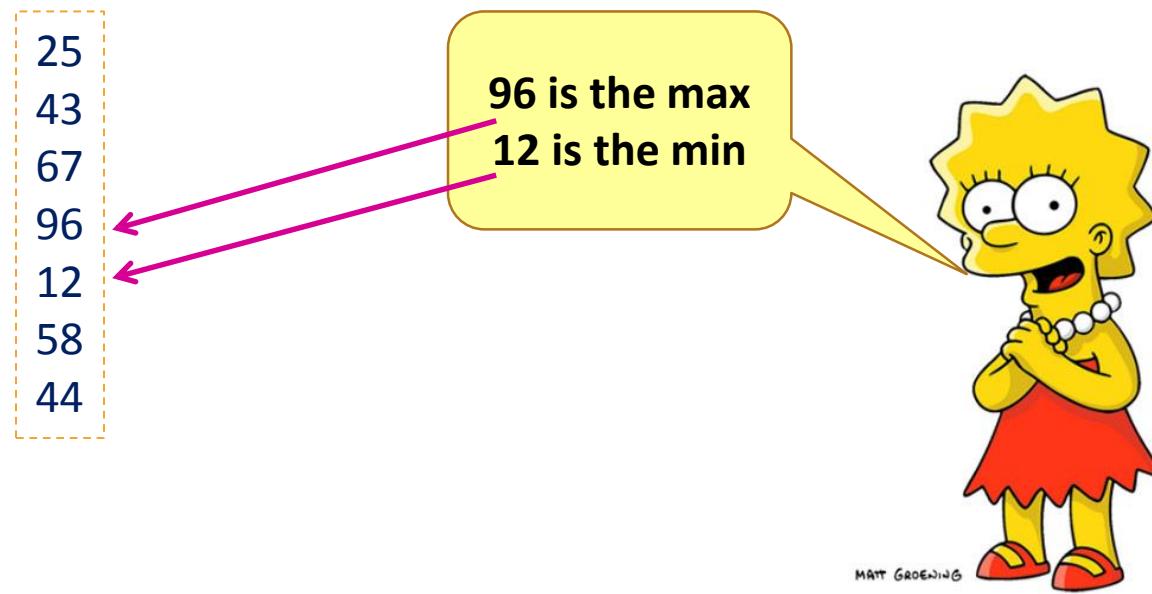
# Searching Improvement

- Stop searching if target has been found

```
scanf("%d", &score);
while (!feof(stdin) && !found) {
    // stop if EOF or target found
    if (score == target)
        found = 1;                  // found = true
    scanf("%d", &score);
}
```

# Looping Subtasks: Finding Extremes

- Finding Extreme Values (e.g. maximum, minimum)
  - Need a variable (such as maxValue) to remember the most extreme value encountered so far



# Looping Subtasks: Finding Extremes

- Finding Extreme Values (e.g. maximum, minimum)
  - Initialize the maxValue (minValue) to some value
    - maxValue: Lower value than any data
    - minValue: Higher value than any data
    - Or for both: The first data value
  - For each data item
    - Compare the current value to maxValue (or minValue)
    - If the current value is > maxValue (< minValue), replace maxValue (minValue) with the current value.

# Extremes Exercise

Write a C program that

1. Reads a set of scores from a file
2. Determines and prints the maximum score

# Looping Subtasks: Finding Extremes

```
// Determine maximum score
int score, maxScore;

scanf("%d", &score);
maxScore = score;           // initialize max

while (!feof(stdin)) {
    if (score > maxScore)
        maxScore = score;    // reset max
    scanf("%d", &score);
}

printf("Maximum score is %d\n", maxScore);
```

# Programming in C



## Chapter 4B Looping Subtasks



THE END

# Programming in C



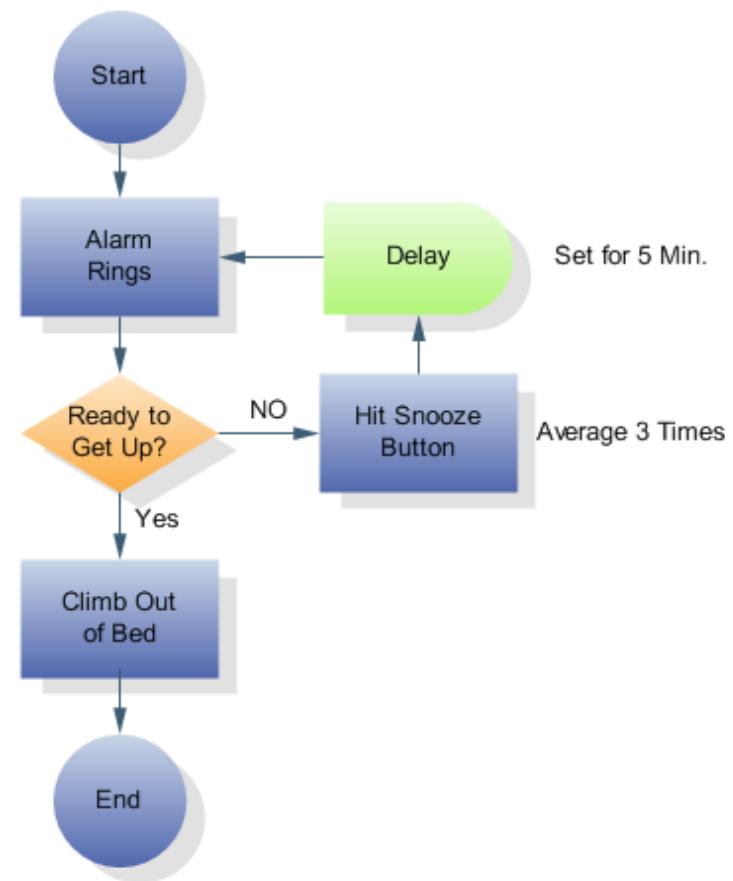
## Chapter 5

### Making Decisions



# Flow of Control

- Flow of control
  - The order in which statements are executed
- Transfer of control
  - When the next statement executed is not the next one in sequence



# Flow of Control

- Control structures

combination of individual statements into a logical unit that regulates the flow of execution in a program or function

- Sequence
- Selection (Making Decisions)
- Repetition (Looping)

# Boolean Expressions

- Evaluate to true or false
- Forms
  - Relational expression: <expr> <relational operator> <expr>
    - Examples:  
`7 < 5`  
`a + b > 6`
  - Logical expression: <Boolean expr> <logical operator> <Boolean expr>
    - Examples:  
`(x < 7) && (y > 3)`

# Relational Operators

Standard Algebraic Relational Operator	C Relational Operator	C Condition Example	Meaning of C Condition
<b>Inequality</b>			
<	<	<code>x &lt; y</code>	<b>x</b> is less than <b>y</b>
$\leq$	$\leq$	<code>x &lt;= y</code>	<b>x</b> is less than or equal to <b>y</b>
>	>	<code>x &gt; y</code>	<b>x</b> is greater than <b>y</b>
$\geq$	$\geq$	<code>x &gt;= y</code>	<b>x</b> is greater than or equal to <b>y</b>
<b>Equality</b>			
=	<code>==</code>	<code>x == y</code>	<b>x</b> is equal to <b>y</b>
$\neq$	<code>!=</code>	<code>x != y</code>	<b>x</b> is not equal to <b>y</b>

4<sup>th</sup>: Ch 4 p. 46  
3<sup>rd</sup>: Ch 5 p. 46

# Logical Operators (Compound Relationals)

Ch 6 p. 72

- **&&** (logical AND)
  - Returns **true** if both conditions are **true**
- **||** (logical OR)
  - Returns **true** if either of its conditions is **true**
- **!** (logical NOT, logical negation)
  - Is a unary operator, only takes one operand following
  - Reverses the truth/falsity of its condition
  - Returns **true** when its condition is **false**

# Logical Operators Truth Table

P	Q	P && Q	P    Q	!P
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

# Precedence of Operators

1. (), []
2. Unary +, unary -, !, ++, --
3. Type casting
4. \*, /, %
5. +, -
6. <, <=, >, >=
7. ==, !=
8. &&
9. ||
10. =

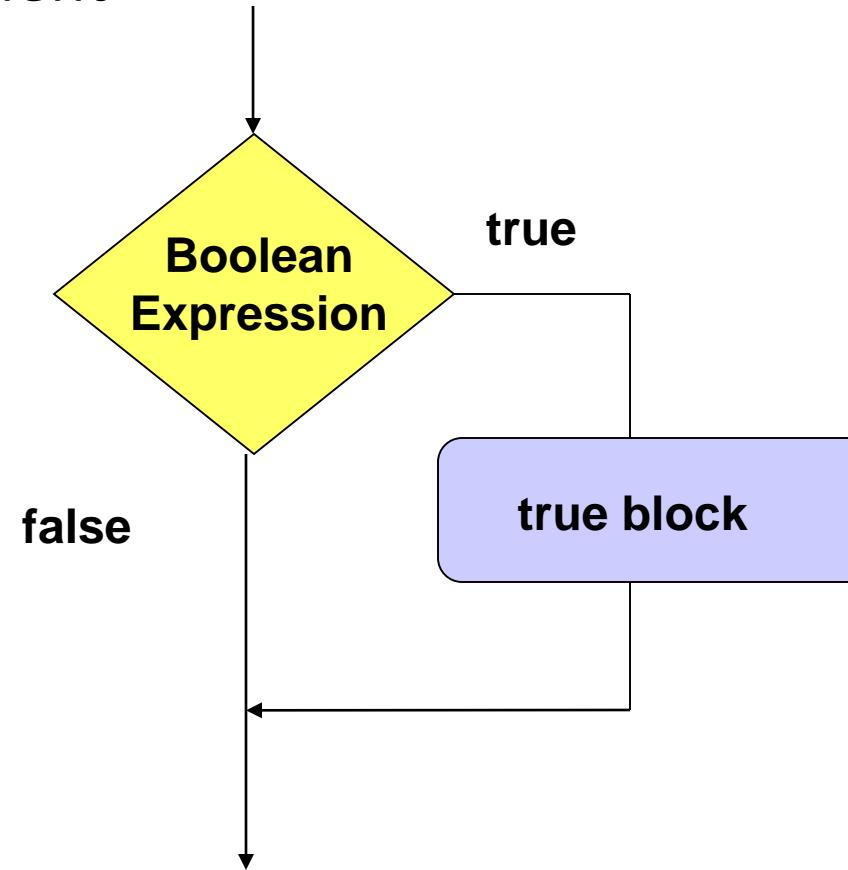
# The *if* Selection Structure

- Selection structure
  - used when we want the computer to choose between two alternative courses of action



# The *if* Selection Structure

- *if* Statement



# The *if* Selection Structure

General form of *if*:

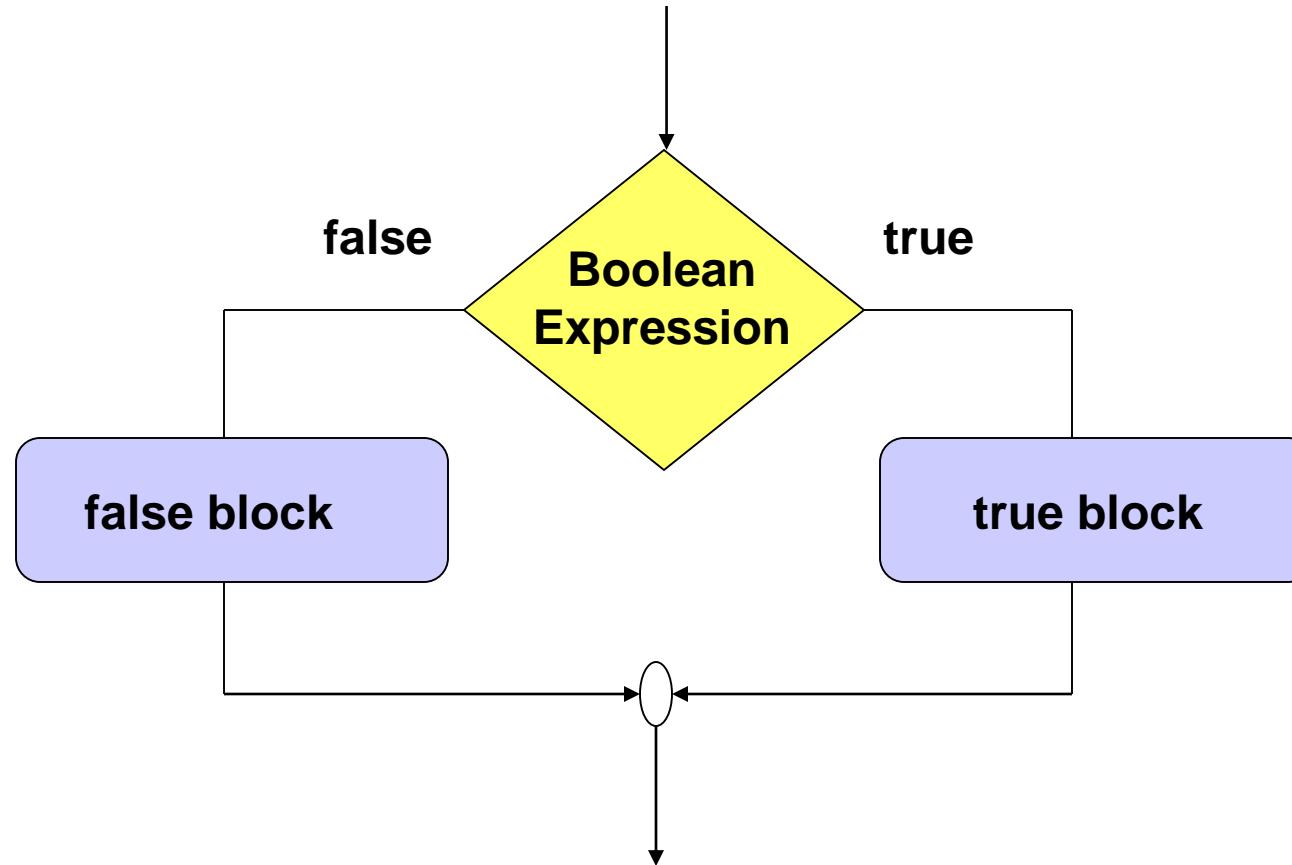
```
if (Boolean Expression)
{
    statement1;
    statement2;
    ...
}
```

# The if-else Selection Structure

- *if*
  - Only performs an action if the condition is true
- *if-else*
  - A different action is performed when condition is true and when condition is false

# *if-else* Selection Structure

*if-else* statement



# The *if-else* Selection Structure

General form of *if-else*:

```
if (expression)
{
    statement1A;
    statement2A;
    ...
}

else
{
    statement1B;
    statement2B;
    ...
}
```

# The *if-else* Selection Structure

- Nested *if-else* structures
  - Test for multiple cases by placing **if-else** selection structures inside **if-else** selection structures.

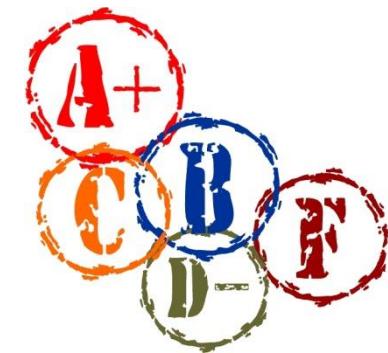


# Nested if-else Structures

```
if (score >= 70)
{
    if (age < 13)
    {
        printf("Great job\n");
    }
    else
    {
        printf("You passed\n");
    }
}
else
{
    printf("You did not pass\n");
}
```

# The *if-else-if* Construct

```
if (grade >= 90)
    printf("A\n");
else
    if (grade >= 80)
        printf("B\n");
    else
        if (grade >= 70)
            printf("C\n");
        else
            if (grade >= 60)
                printf("D\n");
            else
                printf("F\n");
```



- Once a condition is met, the rest of the statements are skipped

# The *if-else-if* Construct

The standard way to indent the previous code is

```
if (grade >= 90)
    printf("A\n");
else if (grade >= 80)
    printf("B\n");
else if (grade >= 70)
    printf("C\n");
else if (grade >= 60)
    printf("D\n");
else
    printf("F\n");
```



Great  
Job!  
A+

# The *if-else* Selection Structure

- Compound statement:
  - Set of statements within a pair of braces
  - Example:

```
if (grade >= 90) {  
    printf("Congratulations!\n");  
    printf("You made an A this course\n");  
}
```



# The *if-else* Selection Structure

- Without the braces, only one statement is executed.  
e.g. given the following code:

```
if (grade >= 90)
    printf("Congratulations!\n");
    printf("You made an A this course\n");
```



- The statement,

```
printf("You made an A this course\n");
```

will be executed independent of the value of grade.

- The statement,

```
printf("Congratulations!\n");
```

will execute only if grade is  
greater than or equal to 90.

# The *dangling* else

```
if (x < y)
    if (x < z)
        printf("Hello\n");
else
    printf("Goodbye\n");
```

**Note:** the compiler matches an else with the closest unmatched if  
The above will be treated as

```
if (x < y)
    if (x < z)
        printf("Hello\n");
else
    printf("Goodbye\n");
```

# The *dangling* else

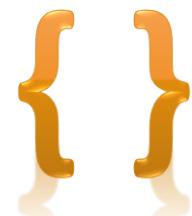
If the else is to match the outer if, use braces.

```
if (x < y)
{
    if (x < z)
        printf("Hello\n");
}
else
    printf("Goodbye\n");
```



# *if-else* Construct

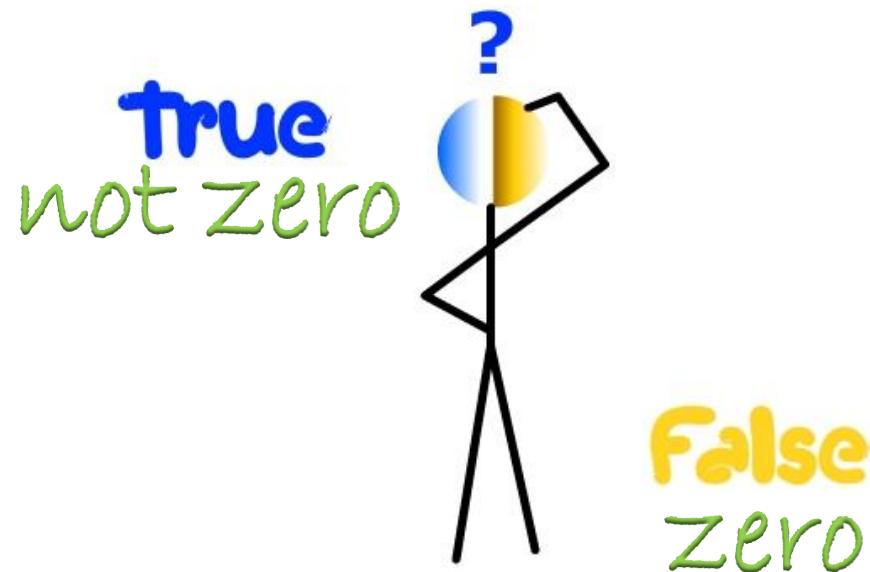
- To avoid confusion, and possible errors, it is best to use braces even for single statements.
  - However, code will be longer



```
if (x < y)
{
    if (x < z)
    {
        printf("Hello\n");
    }
}
else
{
    printf("Goodbye\n");
}
```

# Conditionals

- C uses an integer to represent Boolean values
  - Zero is interpreted as false
  - Any other integer value is interpreted as true



# Conditionals

- `if (n = 0)` is not a syntax error in C.
  - The expression,  $n = 0$ , assigns zero to n and the value of the expression is 0. Zero is interpreted as false, and the false branch of the if statement will be taken.
- `if (n = 5)` is not a syntax error in C.
  - The expression assigns 5 to n. 5 is interpreted as true, and the true branch of the if statement will be taken.

warning: suggest parentheses around assignment used as truth value

# Conditionals



- Remember to use the == operator to test for equality.
- To help catch the error when the equality check involves a constant, put the constant on the left hand side of the ==.
  - For example, use `if (0 == n)`  
instead of `if (n == 0)`

Since `0 = n` is not a valid assignment in C, the compiler will detect this error when == is intended.

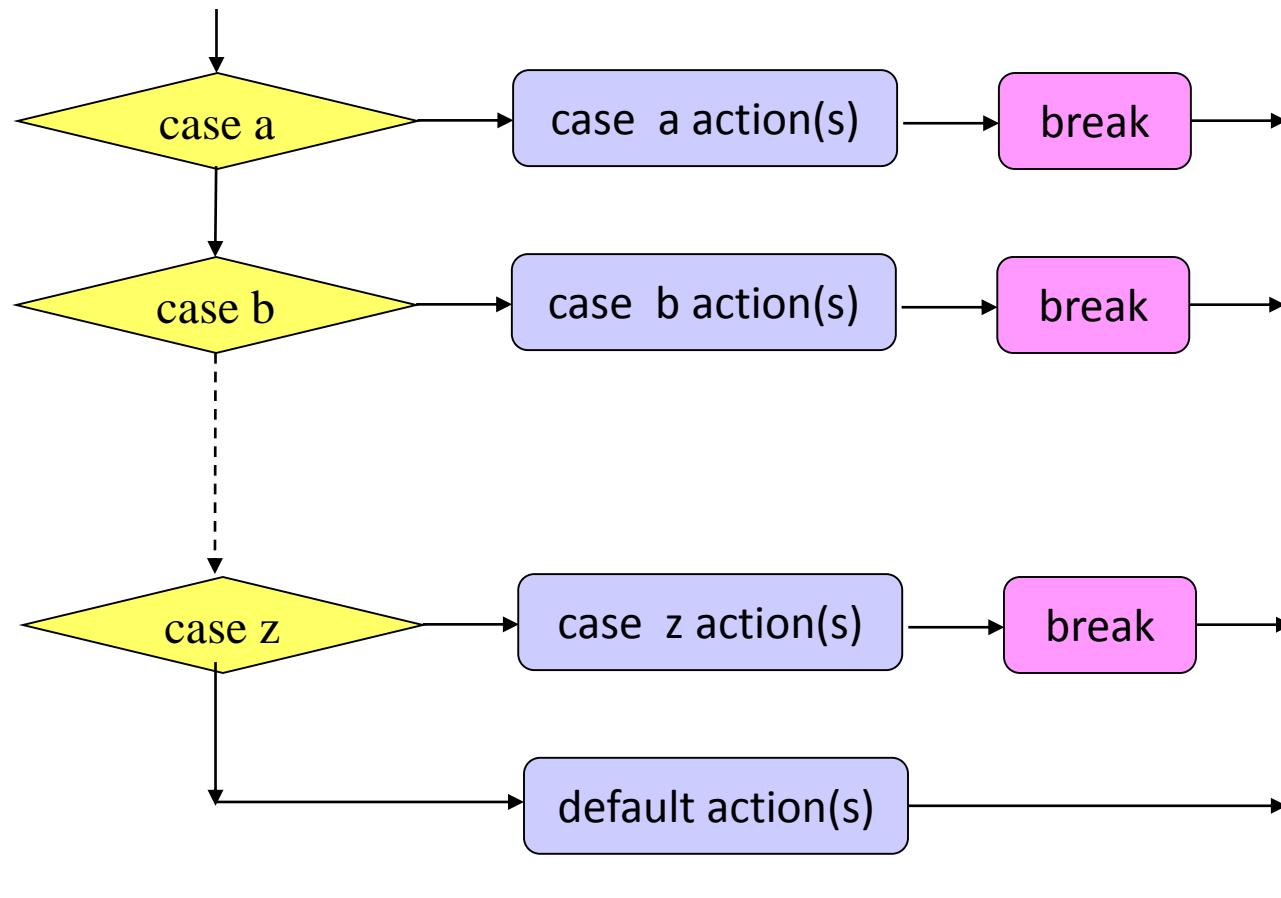
```
error: invalid lvalue in assignment
```

# The *switch* Multiple-Selection Structure

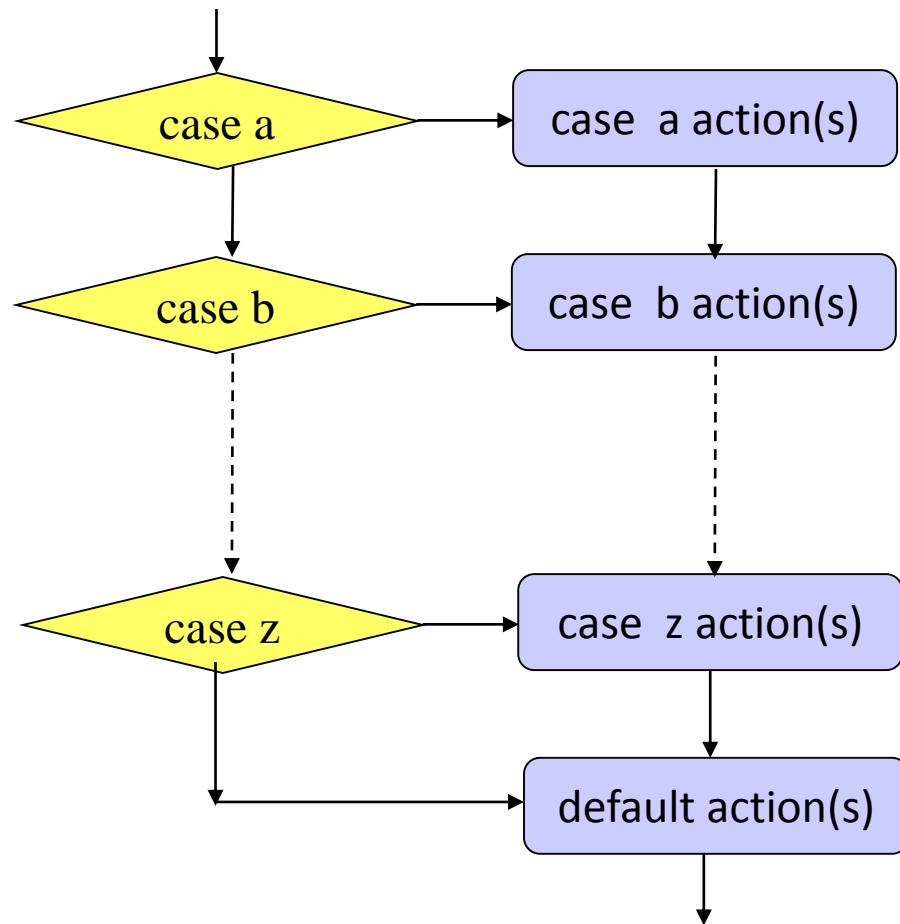
- *switch*
  - Useful when variable or expression is tested for multiple values
  - Consists of a series of **case** labels and an optional **default** case



# The *switch* Multiple-Selection Structure With Breaks

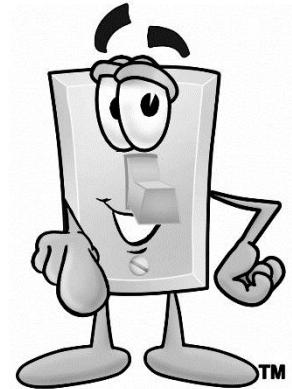


# The *switch* Multiple-Selection Structure Without Breaks



# *switch* Statement Syntax

```
switch (switch_expression)
{
    case constant1:
        statementSequence1
        break;
    case constant2:
        statementSequence2
        break;
    ...
    case constantN:
        statementSequenceN
        break;
    default:
        defaultStmtSequence
}
```





# Switch Statement

- The switch\_expression is compared against the values *constant1, constant2, ..., constantN*
  - *constant1, constant2, ..., constantN* must be simple constants or constant expressions.
    - Can be a char or an int
    - Best to use the same type constant as the switch expression
      - If not, a type conversion will be done.

# *switch* Statement Reminder

- The *switch* statement ends
  - break statement
  - end of the switch statement
- When executing the statements after a case label, it continues to execute until it reaches a break statement or the end of the switch.
- If you omit the break statements, then after executing the code for one case, the computer will continue to execute the code for the next case.



# Example of Switch

```
// Accept letter grade and print corresponding points
printf("Enter letter grade: ");
scanf("%c", &letter_grade);
switch (letter_grade)  {
    case 'A':
    case 'a':
        points = 4.0;
        break;
    case 'B':
    case 'b':
        points = 3.0;
        break;
    case 'C':
    case 'c':
        points = 2.0;
        break;
    case 'D':
    case 'd':
        points = 1.0;
        break;
    case 'F':
    case 'f':
        points = 0.0;
        break;
    default:
        points = 0.0;
        printf("Invalid letter grade\n");
}
```

# Programming in C



## Chapter 5

### Making Decisions

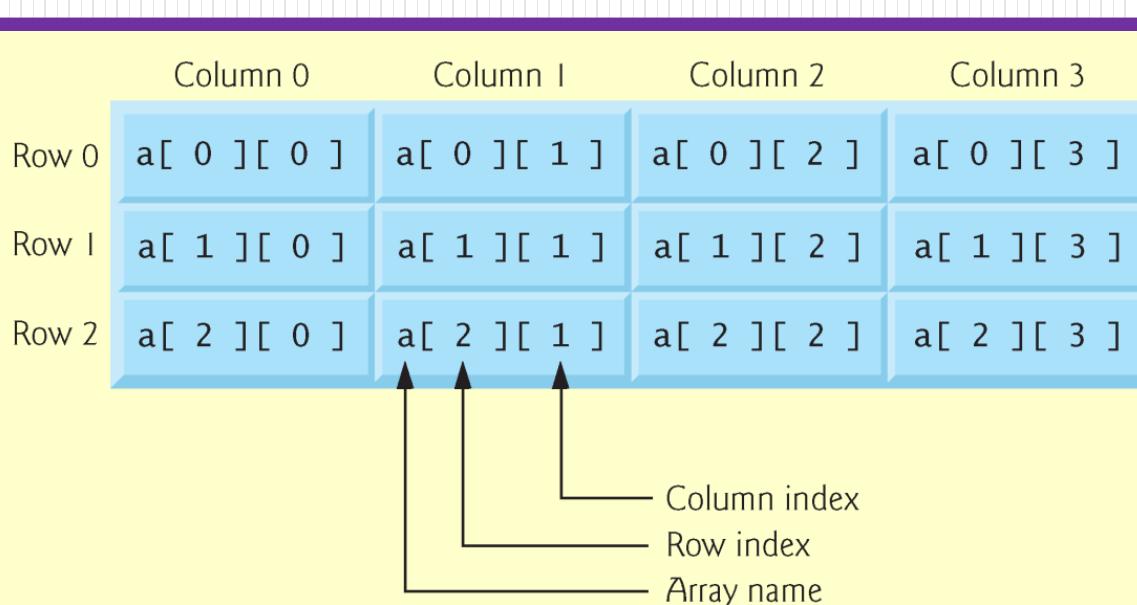
*THE END*

# Programming in C



## Chapter 6A

### Arrays

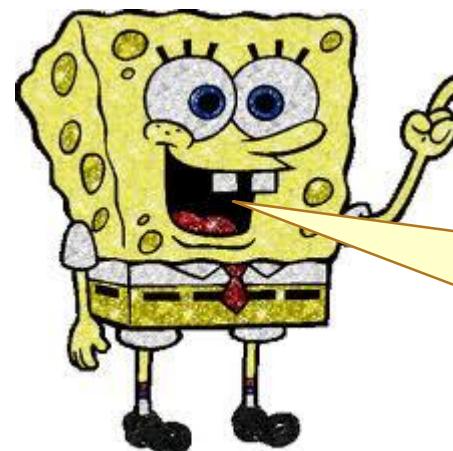


c[ 0 ]	-45
c[ 1 ]	6
c[ 2 ]	0
c[ 3 ]	72
c[ 4 ]	1543
c[ 5 ]	-89
c[ 6 ]	0
c[ 7 ]	62
c[ 8 ]	-3
c[ 9 ]	1
c[ 10 ]	6453
c[ 11 ]	78

# Introduction to Arrays

- A collection of variable data
  - Same name
  - Same type
  - Contiguous block of memory
- Can manipulate or use
  - Individual variables or
  - ‘List’ as one entity

-45
6
0
72
1543
-89
0
62
-3
1
6453
78



Celsius  
temperatures:  
I'll name it c.  
Type is int.

# Introduction to Arrays

- Used for lists of like items
  - Scores, speeds, weights, etc.
  - Avoids declaring multiple simple variables
- Used when we need to keep lots of values in memory
  - Sorting
  - Determining the number of scores above/below the mean
  - Printing values in the reverse order of reading
  - Etc.



# Declaring Arrays

- General Format for declaring arrays

**<data type> <variable> [<size>];**

- Declaration

- Declaring the array → allocates memory
  - Static entity - same size throughout program

- Examples

```
int c[12];
int scores[300];
float weight[3284];
char alphabet[26];
```

Type is int.  
Name is c.

# Defined Constant as Array Size

- Use defined/named constant for array size
  - Improves readability
  - Improves versatility
  - Improves maintainability
- Examples:

```
const int NUMBER_OF_STUDENTS = 50;  
// ...  
int scores[NUMBER_OF_STUDENTS];
```

```
#define NUMBER_OF_STUDENTS 50  
// ...  
int scores[NUMBER_OF_STUDENTS];
```

# Powerful Storage Mechanism

- Can perform subtasks like:
  - "Do this to i-th indexed variable"  
where i is computed by program
  - "Fill elements of array scores from user input"
  - "Display all elements of array scores"
  - "Sort array scores in order"
  - "Determine the sum or average score"
  - "Find highest value in array scores"
  - "Find lowest value in array scores"



# Accessing Array Elements

- Individual parts called many things:
  - Elements of the array
  - Indexed or subscripted variables
- To refer to an element:
  - Array name and subscript or index
  - Format: **arrayname[subscript]**
- Zero based
  - **c[0]** refers to **C<sub>0</sub>**, c sub zero, the **first** element of array **c**

Name of array (note that all elements of this array have the same name, c)

c[ 0 ]	-45
c[ 1 ]	6
c[ 2 ]	0
c[ 3 ]	72
c[ 4 ]	1543
c[ 5 ]	-89
c[ 6 ]	0
c[ 7 ]	62
c[ 8 ]	-3
c[ 9 ]	1
c[ 10 ]	6453
c[ 11 ]	78

Position number of the element within array c

# Accessing Array Elements

- Example

```
printf("%d\n", c[5]);
```

- Note two uses of brackets:

- In declaration, specifies SIZE of array
- Anywhere else, specifies a subscript/index

# Accessing Array Elements

- Example
  - Given the declaration

```
int scores[12];
```

➤ We reference elements of scores by

**scores [0]**

**scores [1]**

...

**scores [11]**

*subscript/index*

```
// Given these element values  
// What does this print?  
printf("%d\n", scores[3]);
```

56
52
80
74
70
95
92
94
80
86
97
87

# Accessing Array Elements

- Size, subscript need not be literal constant
  - Can be named constant or expression

```
int scores[MAX_SCORES]; // MAX_SCORES is a constant  
scores[n+1] = 99;          // If n is 2, same as scores[3]
```



# Major Array Pitfall

- Array indexes go from 0 through size-1!
- C will 'let' you go out of the array's bounds
  - Unpredictable results – may get segmentation fault
  - Compiler will not detect these errors!
- Up to programmer to 'stay in bounds'

```
printf ("%d\n", scores[-8]);  
scores[250] = 88;
```

56
52
80
74
70
95
92
94
80
86
97
87

# for-loops with Arrays

- Natural counting loop
  - Naturally works well 'counting thru' elements of an array
- General form for forward direction
  - `for (subscript = 0; subscript < size; subscript++)`
- General form for reverse direction
  - `for (subscript = size-1; subscript >= 0; subscript--)`

# for-loops with Arrays Examples

```
int scoreSub;  
// Print forward  
for (scoreSub = 0; scoreSub < 12; scoreSub++)  
    printf("Score %d is %d\n", scoreSub+1,  
           scores[scoreSub]);  
  
// Print backward, in reverse  
for (scoreSub = 11; scoreSub >= 0; scoreSub--)  
    printf("Score %d is %d\n", scoreSub+1,  
           scores[scoreSub]);
```

Score 1 is 56  
Score 2 is 52  
Score 3 is 80  
Score 4 is 74  
...  
Score 12 is 87

Score 12 is 87  
Score 11 is 97  
Score 10 is 86  
Score 9 is 80  
...  
Score 1 is 56

56
52
80
74
70
95
92
94
80
86
97
87

# Uses of Defined Constant

- Use everywhere size of array is needed
  - In for-loop for traversal:

```
int score;  
for (score=0; score<NUMBER_OF_STUDENTS; score++)  
    printf("%d\n", scores[score]);
```

- In calculations involving size:

```
lastIndex = NUMBER_OF_STUDENTS - 1;  
lastScore = scores[NUMBER_OF_STUDENTS - 1];
```

- When passing array a function:

```
total = sum_scores(scores, NUMBER_OF_STUDENTS);
```

# Array as Function Parameter

- Include type and brackets []
  - Size inside brackets is optional and is ignored
- Passes pointer/reference to array
  - Function can modify array elements
- Common to also pass size
- Example:

```
void print_scores(int values[], int num_values) {  
    // Call: print_scores(scores, scoreCount)  
    int valueNdx;  
    for (valueNdx=0; valueNdx<num_values; valueNdx++)  
        printf("%d\n", values[valueNdx]);  
}
```

# Initializing Arrays

..... INit

- Arrays can be initialized at declaration

```
int scores[3] = {76, 98, 83};
```

- Size cannot be variable or named constant
- Equivalent to

```
int scores[3];
scores[0] = 76;
scores[1] = 98;
scores[2] = 83;
```

# Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills 'rest' with zero of array base type
    - Declaration

```
int scores[5] = {76, 98, 83}
```

- Performs initialization

```
scores[0] = 76;  
scores[1] = 98;  
scores[2] = 83;  
scores[3] = 0;  
scores[4] = 0;
```



# Auto-Initializing Arrays

- If array size is left out
  - Declares array with size required based on number of initialization values
  - Example:

```
int scores[] = {76, 98, 83}
```
  - Allocates array scores with size of 3



# Multidimensional Arrays

- Arrays with more than one dimension
  - Declaration: Additional sizes each enclosed in brackets
- Two dimensions
  - Table or ‘array of arrays’

```
int a[3][4];
```

- Requires two subscripts – row and column

	Column 0	Column 1	Column 2	Column 3
Row 0	a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
Row 1	a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
Row 2	a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

Diagram illustrating a 3x4 multidimensional array:

- The array is represented as a grid of 12 cells.
- It has 3 rows labeled Row 0, Row 1, and Row 2.
- It has 4 columns labeled Column 0, Column 1, Column 2, and Column 3.
- Each cell contains a reference to the array element: a[ row ][ column ].
- A coordinate system is shown at the bottom left:
  - The vertical axis is labeled "Row index".
  - The horizontal axis is labeled "Column index".
  - The origin is labeled "Array name".



# Initializing Multidimensional

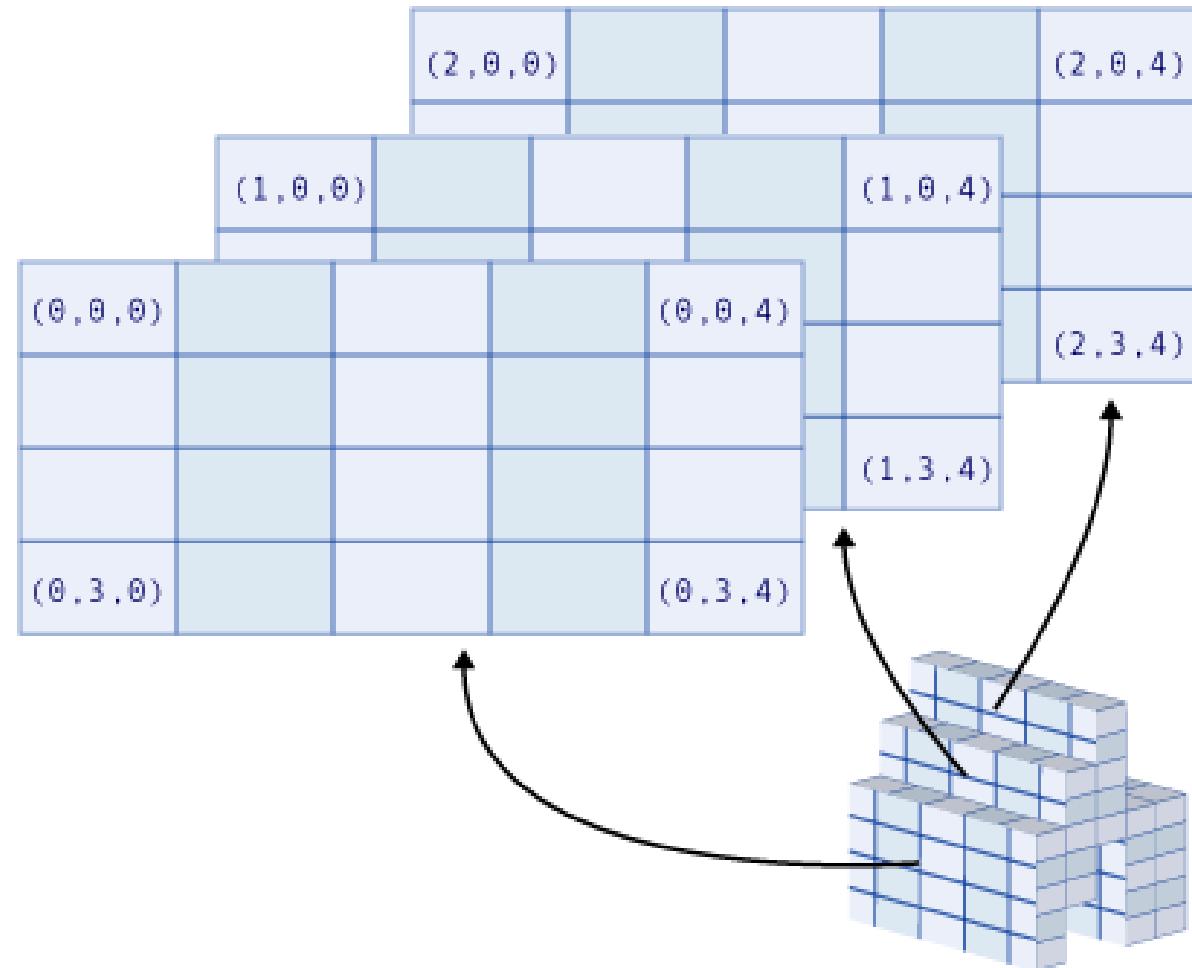
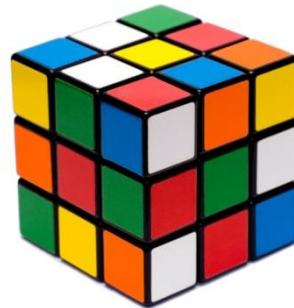


- Nested lists
  - Unspecified values set to zero
- 2D Example:

```
int nums[4][5] = { {10, 6, -7, 13, 28},  
                   {10, 5, 44, 8},  
                   {33, 20, 1, 0, 14},  
                   { 2, 66, 25, 37, 1}  
}
```

# Three-dimensional Visualization

```
int cube[3][3][3];
```



# Multidimensional Array Parameters

- Must specify size after first dimension

```
void scalar_multiply(int rows, int cols,
                     int a[][cols], int scalar) {
    // multiplies each element in array by scalar
    int row, col;
    for (row=0; row<rows; row++)
        for (col=0; col<cols; col++)
            a[row][col] *= scalar;
}
```

# Programming in C



## Chapter 6A

### Arrays

*THE END*



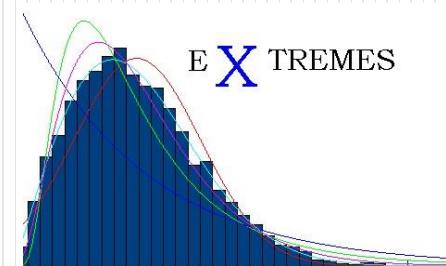
# Programming in C



## Chapter 6B Array Subtasks



-45		-89
6		-45
0		-3
72		0
1543		0
-89	A	1
0	Z	6
62		62
-3		72
1		78
6453		1543
78		6453

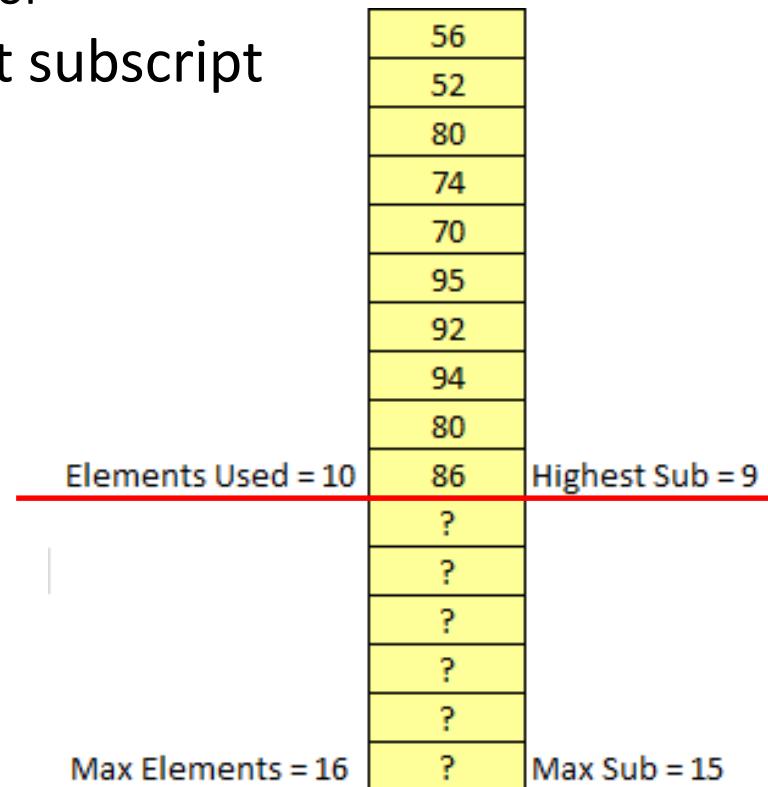
 $\Sigma$ 

# Programming with Arrays

- Subtasks
  - Partially-filled arrays
  - Loading
  - Searching
  - Sorting
  - Sum, average
  - Extremes

# Partially-filled Arrays (Common Case)

- Must be declared some maximum size
- Program must maintain
  - How many elements are being used  
and/or
  - Highest subscript





# Sizeof and Arrays

- Operator sizeof returns the total bytes in the argument
  - Total elements = sizeof(array) / sizeof(data-type)
  - ```
int scores[MAX_SCORES];
int scoresBytes = sizeof(scores);    // MAX_SCORES * 4
int scoresElements = sizeof(scores) / sizeof(int); // MAX_SCORES
```
- Sizeof does not return total bytes being used
  - ! You cannot use sizeof to determine the number of elements being used in a partially filled array



# Loading an Array



- Be careful not to overfill
  - Do not read directly into array elements

```
// Example: Load array of scores checking for overfill
const int MAX_SCORES = 50;
int scores[MAX_SCORES];
int score, scoreCount;

// Load into array, check for too many
for (scoreCount=0; scanf("%d", &score) == 1; scoreCount++) {
    // scoreCount here is one less than actual scores read
    if (scoreCount >= MAX_SCORES) {
        printf("Unable to store more than %d scores.\n", MAX_SCORES);
        exit(1);      // stdlib: exits program even in nested function
    }
    scores[scoreCount] = score;
}
```

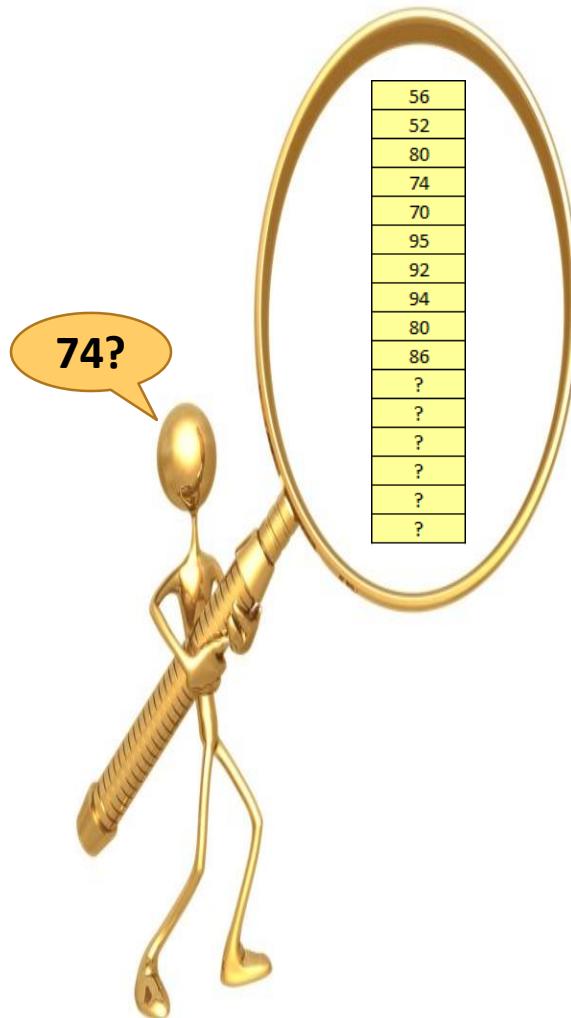
# Loading a Two-dimensional Array

```
void load_table(int rows, int cols, int a[][][cols]) {  
    // assumes data matches table dimensions  
    int row, col, value;  
    for (row=0; row<rows; row++)  
        for (col=0; col<cols; col++) {  
            scanf("%d", &value);  
            a[row][col] = value;  
        }  
}
```

# Safer 2D Load

```
int load_table(int rows, int cols, int a[][][cols]) {
    // verifies table matches data
    // returns 1 if match, otherwise 0
    int row, col, value;
    int match = 1;
    scanf("%d", &value);
    for (row=0; !feof(stdin) && row<rows; row++) {
        for (col=0; !feof(stdin) && col<cols; col++) {
            a[row][col] = value;
            scanf("%d", &value);
        }
        // if !feof(stdin) then too much data in file
        // if row!=rows then not enough data in file
        if (!feof(stdin) || row!=rows)
            match = 0;
    return match;
}
```

# Searching an Array



- Linear search
  - Simple
- Binary search
  - Requires sorted array
  - Generally faster for large arrays
- May require the use of an indicator to denote found or not found

```
// Target found indicator  
int found = 0;
```

# Linear Search Example Using While

```
// Example: Search array using while
int scores[MAX_SCORES];
int scoreCount, scoreNdx, targetScore;

// Assume array has been loaded,
// count = scoreCount, and search value = targetScore
scoreNdx = 0;
while (scoreNdx<scoreCount && scores[scoreNdx] !=targetScore)
    scoreNdx++;
if (scoreNdx>=scoreCount) {
    // Whatever you want to do if not found
}
else {
    // Whatever you want to do if found
}
```

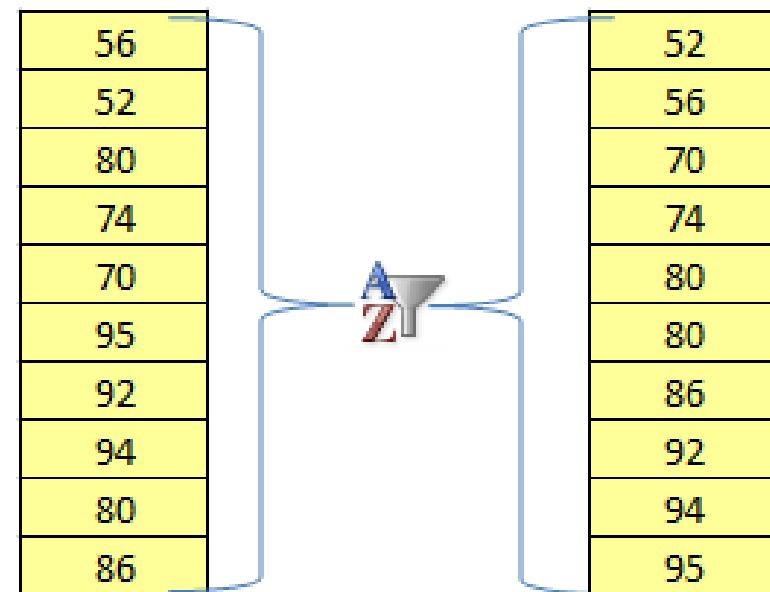
# Linear Search Example Using For

```
// Example: Search array using for
int scores[MAX_SCORES];
int scoreCount, scoreNdx, targetScore;

// Assume array has been loaded,
// count = scoreCount, and search value = targetScore
for (scoreNdx=0;
     scoreNdx<scoreCount && scores[scoreNdx]!=targetScore;
     scoreNdx++) /* null */;
// Note: Above for statement has empty basic block by design
if (scoreNdx>=scoreCount) {
    // Whatever you want to do if not found
}
else {
    // Whatever you want to do if found
}
```

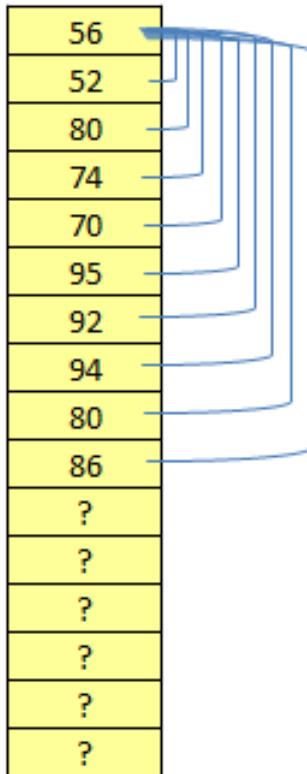
# Sorting

- Place array into some order
  - Ascending or descending
- Many types
  - Simple: Selection
  - More intelligent: Bubble, selection, insertion, shell, comb, merge, heap, quick, counting, bucket, radix, distribution, timsort, gnome, cocktail, library, cycle, binary tree, bogo, pigeonhole, spread, bead, pancake, ...



# Selection Sort

- Compare element to all elements below and then move to next element, swap when appropriate



```
void sort_values(int values[], int count) {
    // Sort values in ascending order
    // using selection sort
    int sub1, sub2, temp;

    for (sub1=0; sub1<count-1; sub1++)
        for (sub2=sub1+1; sub2<count; sub2++)
            if (values[sub1] > values[sub2]) {
                temp = values[sub1]; // swap
                values[sub1] = values[sub2];
                values[sub2] = temp;
            }
}
```

# Bubble/Sinking Sort

- Compare adjacent elements, swap when appropriate
- Stop if no swaps on a pass

|    |
|----|
| 56 |
| 52 |
| 80 |
| 74 |
| 70 |
| 95 |
| 92 |
| 94 |
| 80 |
| 86 |
| ?  |
| ?  |
| ?  |
| ?  |
| ?  |
| ?  |

```
void sort_values(int values[], int count) {
    // Sort values in ascending order
    // using bubble sort
    int sub1, sub2, temp, sorted = 0;

    for (sub1=0; !sorted && sub1<count-1; sub1++) {
        sorted = 1;      // Assume sorted on each pass
        for (sub2=count-2; sub2>=sub1; sub2--)
            if (values[sub2] > values[sub2+1]){
                temp = values[sub2]; // swap
                values[sub2] = values[sub2+1];
                values[sub2+1] = temp;
                sorted = 0;      // Assume unsorted after swap
            }
    }
}
```

# Sum & Average Example

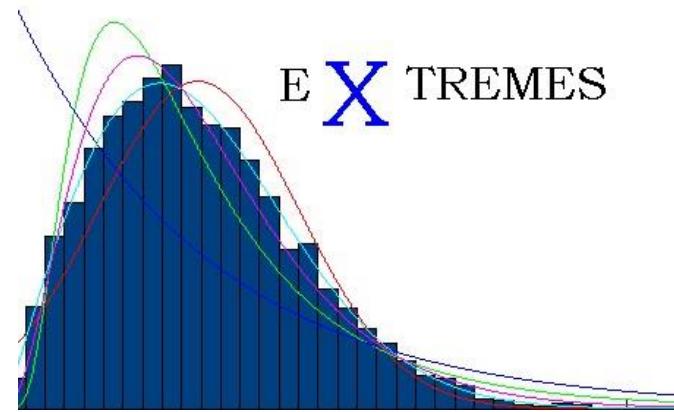
- Verify positive count before computing average
  - Protects against division by zero

```
// Calculate average score
int scores[MAX_SCORES];
int scoreCount, scoreNdx, sum;
float average;

// Assume array has been loaded, count = scoreCount
if (scoreCount <= 0) // Verify positive count
    printf("Unable to compute average, no scores\n");
else {
    sum = 0;
    for (scoreNdx=0; scoreNdx<scoreCount;
         scoreNdx++)
        sum+= scores[scoreNdx];
    average = (float) sum / scoreCount;
    printf("Average score is %.2f\n", average);
}
```

# Extremes

- Same techniques as chapter 5 – best:
  - Assume first is extreme
  - Compare others to current extreme
  - Replace extreme when finding new extreme



# Extremes: Find Maximum Example

```
int scores[MAX_SCORES];
int scoreCount, scoreNdx, maxScore;

// Assume array has been loaded, count = scoreCount
maxScore = scores[0];    // Assume first
for (scoreNdx=1; scoreNdx<MAX_SCORES; scoreNdx++)
    if (scores[scoreNdx] > maxScore) // Check others
        maxScore = scores[scoreNdx];
printf("The highest score is %d\n", maxScore);
```

# Programming in C



## Chapter 6B Array Subtasks

*THE END*

# Programming in C



## Chapter 7

### Programmer-Defined Functions

main

Level 2

Level 2

Level 2

Level 3

Level 3



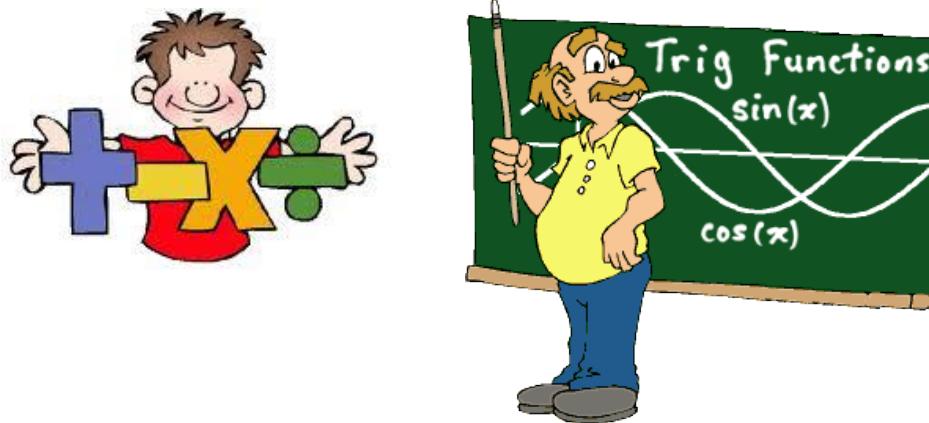
# Programmer-Defined Functions

- Modularize with building blocks of programs
  - Divide and Conquer
    - Construct a program from smaller pieces or components
      - Place smaller pieces into functions
    - Pieces are more manageable than one big program
      - Makes other functions smaller
      - Pieces can be independently implemented and tested



# Programmer-Defined Functions

- Readability
  - Function name should indicate operations performed
- Reuse
  - Functions may be used multiple times in same program
  - Functions may be used in other programs



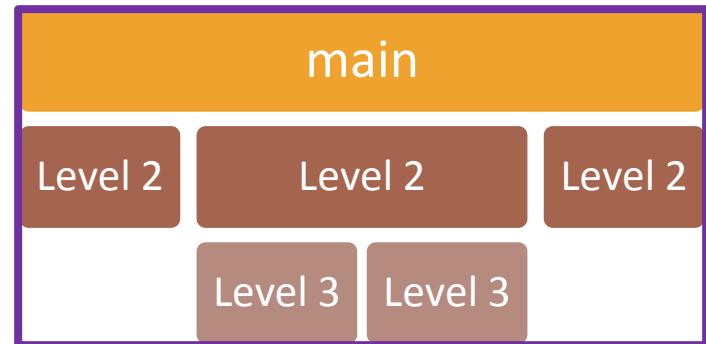
# Components of Function Use

- Three steps to implementing functions
  1. Function declaration/prototype
    - If not defined before use
  2. Function definition
  3. Function call
    - Either prototype or definition must come first
- Prototype and/or definitions can go in either
  - Same file as main()
  - Separate file so other programs can also use it
    - #include

1.2.3

# Program Function Definition Structure

- main first (preferred)
  - Top down design
  - Some prototypes required
  - Complete prototyping allows function definition in any order
- main is last – lowest level functions first
  - Bottom up design
  - Prototypes not required
- main in the middle
  - Confusing: **Do not do!**



# 1. Function Declaration/Prototype

- An ‘informational’ declaration for compiler
- Tells compiler how to interpret calls
- Syntax:

`<return_type> FnName(<formal-parameter-list>);`

- Formal parameter syntax:

`<data_type> Parameter-Name`

- Example:

`char grade(int score);`

# Function Declaration/Prototype

- Placed before any calls
  - Generally above all functions in global space
  - May be placed in declaration space of calling function
- Example

```
#include <stdio.h>

// Function prototypes
double total_cost(int quantity, double unit_cost);

int main() {
```

# Alternative Function Declaration

- Function declaration is 'information' for compiler, so
  - Compiler only needs to know:
    - Return type
    - Function name
    - Parameter list
      - Formal parameter names not needed but help readability
- Example

```
#include <stdio.h>

// Function prototypes
double total_cost(int, double);

int main() {
```

## 2. Function Definition

- Actual implementation/code for what function does
  - Just like implementing function main()
  - General format – header & basic block:  
`<return-type> fn-name (parameter-list)`        
`basic block`
- Example:

```
double total_cost(int quantity, double unit_cost) {
    const double TAXRATE = 0.05;
    double sub_total;
    sub_total = quantity * unit_cost;
    return (sub_total + sub_total * TAXRATE);
}
```

# Return Statements

- Syntax: **return return-value-expression**
- Two actions
  - Sets return value
  - Transfers control back to 'calling' function
- **Good programming & course requirement:**
  - One return per function
  - Return is last statement

```
double total_cost(int quantity, double unit_cost) {  
    const double TAXRATE = 0.05;  
    double sub_total;  
    sub_total = quantity * unit_cost;  
    return (sub_total + sub_total * TAXRATE);  
}
```



### 3. Function Call

- Using function name transfers control to function
  1. Values are passed through parameters
  2. Statements within function are executed
  3. Control continues after the call
- For value-returning functions, either
  - Store the value for later use

```
bill = total_cost(number, price);
```
  - Use the value returned without storing

```
printf("Cost is %f\n", total_cost(number, price));
```
  - Throw away return value

```
total_cost(number, price);
```

# Parameters (Arguments)

- Formal parameters/arguments
  - In function declaration
  - In function definition's header
  - 'Placeholders' for data sent in
  - 'Variable name' used to refer to data in definition of function
- Actual parameters/arguments
  - In function call



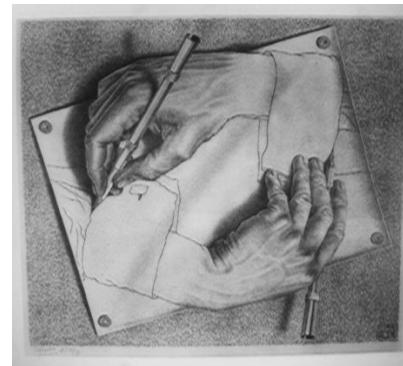
# Parameter vs. Argument

- Names used interchangeably
- Technically parameter is 'formal' piece while argument is 'actual' piece



# Functions Calling Functions

- We're already doing this!
  - main() IS a function calling printf!
- Only requirement:
  - Function's declaration or definition must appear first
- Common for functions to call many other functions
  - Function can call itself → Recursion

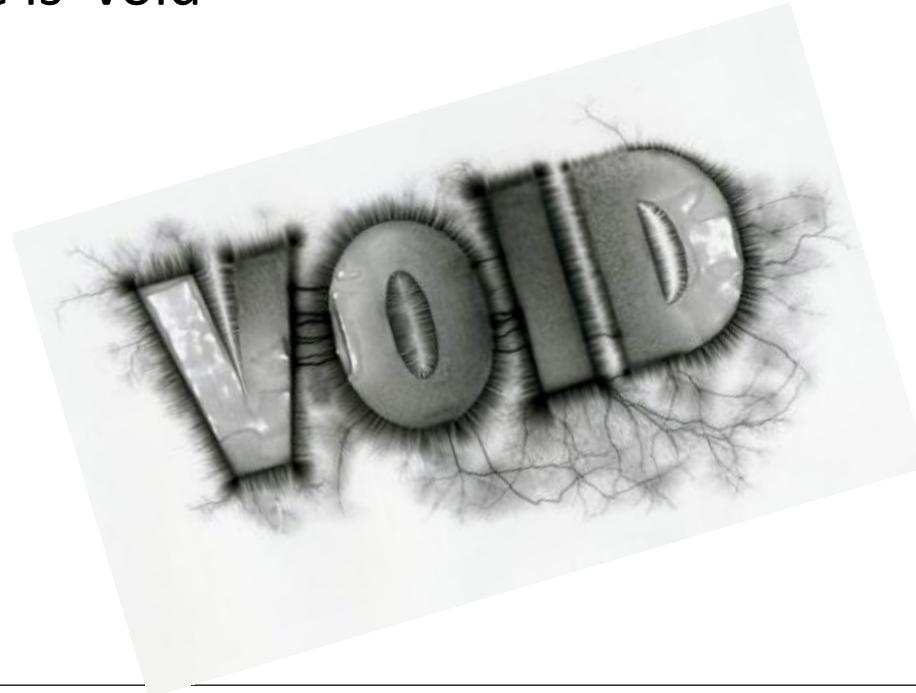


# Declaring Void Functions

- Similar to functions returning a value
  - Return type specified as 'void'
- Example prototype:

```
void showResults(double fDegrees, double cDegrees);
```

- Return-type is 'void'



# Declaring Void Functions

- Nothing is returned
  - Void functions cannot have return statement with an expression
    - Will return at end of function
  - Non-void functions must have return statement with an expression
- Example definition:

```
void showResults(double fDegrees, double cDegrees) {  
    printf("%.2f degrees fahrenheit equals ", fDegrees);  
    printf("%.2f degrees celsius\n", cDegrees);  
}
```

# Calling Void Functions

- From some other function, like main():

```
showResults (degreesF, degreesC) ;  
showResults (32.5, 0.3) ;
```

- Cannot be used where a value is required
  - Cannot be assigned to a variable, since no value returned

# Function documentation

- Used to aid in program maintenance
- Comments at non-main definition header
  - Purpose of function
  - Parameters
  - Return
  - **Class standard example:**



```
double interest(double balance, double rate);
// Calculates the interest charge on an account balance
// Parameters:    balance - non-negative account balance
//                           rate - interest rate percentage
// Return:          calculated interest charge
```

# main(): 'Special'

- Recall: main() IS a function
- 'Special'
  - It is the first function executed
  - Called by operating system or run-time system
  - Can return value to operating system
    - Value can be tested in command scripts
- Tradition holds it should return an int
  - Zero indicates normal ending of program

# Scope of Identifier Names



- Region of a program where identifier is visible
  - Begins at definition within block
  - Ends at end of block
- Local variables
  - Name given to variables defined within function block
  - Can have different local variables with same name declared in different functions
  - Cannot have duplicate local names within a function

# Scope Rules

- Local variables preferred
  - Maintain individual control over data
  - Need to know basis (Hidden)
  - Functions should declare whatever local data needed to 'do their job'





# Global Scope

- Names declared 'outside' function bodies
  - Global to all functions in that file
- Global declarations typical for constants:
  - Declare globally so all functions have scope, can use

```
#include <stdio.h>

const double TAX_RATE = 0.05;

int main() {
```

# Global Constants and Global Variables

- Global variables?
  - Possible, but SELDOM-USED
  - Better alternative is to use parameters
  - Dangerous: no control over usage!
  - **We do not use global variables in this class!**



# Block Scope

- Declare data inside nested blocks
  - Has 'block-scope'
    - Note: All function definitions are blocks!

```
if (amount > 5) {  
    int add_in;  
    add_in = prior_amount * .05;  
    amount += add_in;  
}
```



# Lifetime

- How long does it last
  - Allocation  $\Rightarrow$  Deallocation
- Normally variables are allocated when defined
- Normally variables are deallocated at the end of block

```
double total_cost(int quantity, double unit_cost) {  
    const double TAXRATE = 0.05; // TAXRATE allocated  
    double sub_total;           // sub_total allocated  
    sub_total = quantity * unit_cost;  
    return (sub_total + sub_total * TAXRATE);  
} // TAXRATE and sub_total deallocated
```

# Static & Lifetime

- Variable definition modifier keyword: **static**
- Static variables are only allocated once
- Static variables are not deallocated until program ends

```
int keep_count() {  
    static int count = 0;  
    // count will remain allocated and keep its value  
    count++;  
    return count;  
}
```

# Programming in C



## Chapter 7

### Programmer-Defined Functions

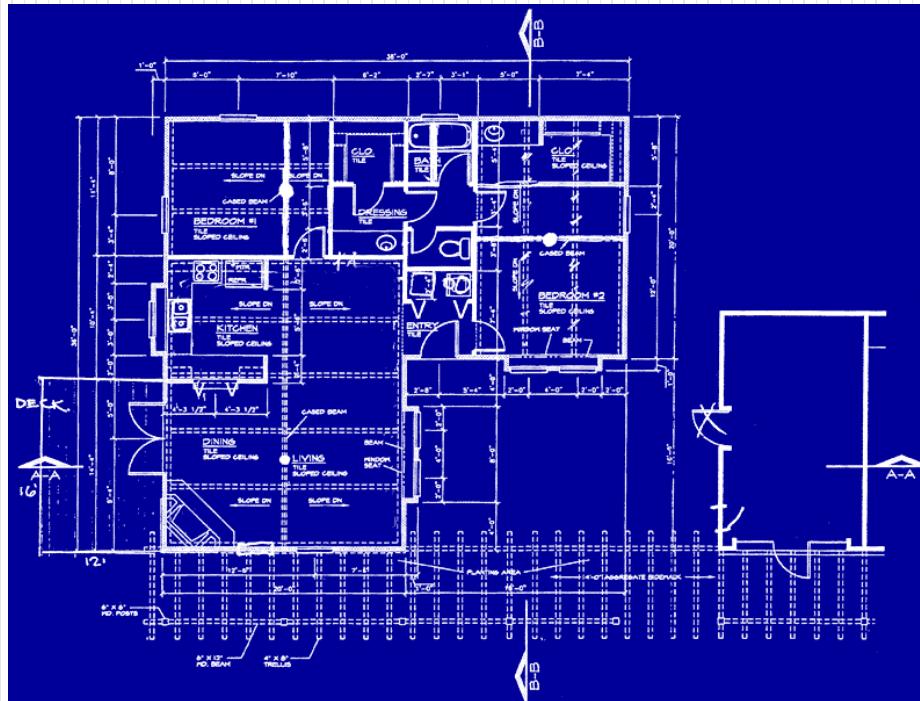
*THE END*

# Programming in C



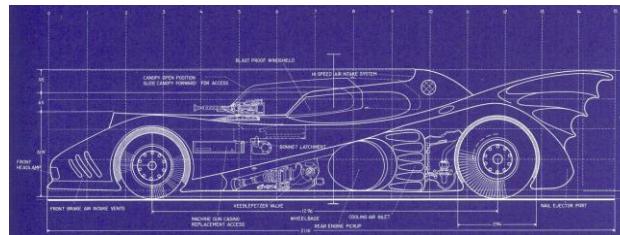
## Chapter 8

### Structures



# Structures

- A structure can be used to define a new data type that combines different types into a single (compound) data type
  - Definition is similar to a template or blueprint
  - Composed of members of previously defined types



- Structures must be defined before use
- C has three different methods to define a structure
  - variable structures
  - tagged structures
  - type-defined structures

# 1) Struct variable

- A variable structure definition defines a struct variable

```
struct {  
    double x; // x coordinate  
    double y; // y coordinate  
} point;
```

Variable name

*DON'T FORGET THE SEMICOLON*

Member names

## 2) Tagged Structure

- A tagged structure definition defines a type
- We can use the tag to define variables, parameters, and return types

```
struct point_t {  
    double x; // x coordinate  
    double y; // y coordinate  
};
```

Structure tag

Member names

*DON'T FORGET THE SEMICOLON*

- Variable definitions:

```
struct point_t point1, point2, point3;
```

- Variables point1, point2, and point3 all have members x and y.

### 3) Typedef Structure

- A typed-defined structure allows the definition of variables without the struct keyword.
- We can use the tag to define variables, parameters, and return types.

```
typedef struct {  
    long ssn;          // Social Security Number  
    int empType;       // Employee Type  
    float salary;      // Annual Salary  
} employee_t;
```

New type name

DON'T FORGET THE SEMICOLON

Member names

- Variable definition:

```
employee_t emp;
```

- Variable emp has members ssn, empType, and salary.



# Dot Operator (.)

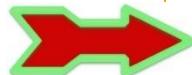
- Used to access member variables
  - Syntax:  
`structure_variable_name.member_name`
  - These variables may be used like any other variables

```
struct point_t {  
    double x; // x coordinate  
    double y; // y coordinate  
};  
void setPoints() {  
    struct point_t point1, point2;  
    point1.x = 7;    // Init point1 members  
    point1.y = 11;  
    point2 = point1; // Copy point1 to point2  
    ...  
}
```

# Arrow Operator (->)

- Used to access member variables using a pointer
  - Arrow Operator Syntax:  
`structure_variable_pointer->member_name`
  - Dot Operator Syntax:  
`(*structure_variable_pointer).member_name`

```
typedef struct {  
    long ssn;          // Social Security Number  
    int empType;       // Employee Type  
    float salary;      // Annual Salary  
} employee_t;  
  
employee_t * newEmp(long n, int type, float sal) {  
    employee_t * empPtr = malloc(sizeof(employee_t));  
    empPtr->ssn = n;                      // -> operator  
    empPtr->empType = type;                // -> operator  
    (*empPtr).salary = sal;                 // dot operator  
    return empPtr;  
}
```





# Nested Structures

- A member that is of a structure type is nested

```
typedef struct {  
    int month;  
    int day;  
    int year;  
} date_t;  
  
typedef struct {  
    double height;  
    int weight;  
    date_t birthday;  
} personInfo_t;  
  
// Define variable of type personInfo_t  
personInfo_t person;  
...  
  
// person.birthday is a member of person  
// person.birthday.year is a member of person.birthday  
printf("Birth year is %d\n", person.birthday.year);
```



Initializing...

# Initializing Structures

- A structure may be initialized at the time it is declared
- Order is essential
  - The sequence of values is used to initialize the successive variables in the struct
- It is an error to have more initializers than members
- If fewer initializers than members, the initializers provided are used to initialize the data members
  - The remainder are initialized to 0 for primitive types

```
typedef struct {  
    int month;  
    int day;  
    int year;  
} date_t;
```

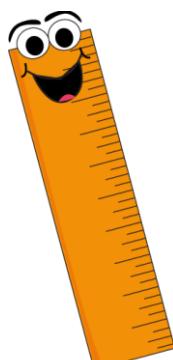
```
date_t due_date = {12, 31, 2020};
```

# Dynamic Allocation of Structures

- The `sizeof()` operator should always be used in dynamic allocation of storage for structured data types and in reading and writing structured data types

```
typedef struct {  
    int month;  
    int day;  
    int year;  
} date_t;  
  
date_t due_date;  
  
int date_t_len = sizeof(date_t); // sizeof type  
int due_date_len = sizeof(due_date); // sizeof variable  
  
printf("sizeof(date_t)=%d\n", date_t_len);  
printf("sizeof(due_date)=%d\n", due_date_len);  
  
date_t * due_dates = calloc(100, sizeof(date_t));
```

sizeof(date\_t)=12  
sizeof(due\_date)=12



|   |
|---|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

# Arrays Within Structures

- A member of a structure may be an array

```
typedef struct {
    long ssn;                      // SSN
    double payRate;                 // Hourly rate
    float hoursWorked[7];          // Daily hours worked Sun-Sat
} timeCard_t;

timeCard_t empTime;

empTime.hoursWorked[5] = 6.5;    // Thur hours worked
```

# Arrays of Structures

- We can also create an array of structure types

```
typedef struct {
    // unsigned char will hold 0-255
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} pixel_t;

pixel_t pixelMap[800][600];

pixelMap[425][37].red = 127;
pixelMap[425][37].green = 0;
pixelMap[425][37].blue = 58;
```

# Arrays of Structures Containing Arrays

- We can also create an array of structures that contain arrays

```
typedef struct {
    long ssn;                  // SSN
    double payRate;             // Hourly rate
    float hoursWorked[7];      // Daily hours worked Sun-Sat
} timeCard_t;

timeCard_t empTime[1000];

// Thur hours worked, emp # 10

empTime[9].hoursWorked[5] = 6.5;
```

# Structures as Parameters

- A struct, like an int, may be passed to a function
- The process works just like passing an int, in that:
  - The complete structure is copied to the stack
  - Called function is unable to modify the caller's copy of the variable





# Structures as Parameters

```
typedef struct {
    double x; // x coordinate
    double y; // y coordinate
} point_t;

void changePoint(point_t p) {
    printf("x=%.1lf, y=%.1lf\n", p.x, p.y);
    //
    p.x = 3.4;
    p.y = 4.5;
}

void mainPoint() {
    point_t point = {1.2, 2.3};
    changePoint(point);
    printf("x=%.1lf, y=%.1lf\n", point.x, point.y);
    //
}
```

x=1.2, y=2.3

x=1.2, y=2.3



# Structures as Parameters

- Disadvantage of passing structures by value:  
Copying large structures onto stack
  - Is inefficient
  - May cause stack overflow

```
typedef struct {
    int w[1000*1000*1000]; // One billion int elements
} big_t;

// Passing a variable of type big_t will cause
// 4 billion bytes to be copied on the stack

big_t fourGB;

int i;
for (i = 0; i < 1000000; i++) // 1,000,000 times
    slow_call(fourGB);
```

# Structure Pointers as Parameters

- More efficient: Pass the address of the struct
- Passing an address requires that only a single word be pushed on the stack, no matter the size
  - Called function can then modify the structure.





# Structure Pointers as Parameters

```
typedef struct {
    double x; // x coordinate
    double y; // y coordinate
} point_t;

void changePoint(point_t * p) {
    printf("x=%lf, y=%lf\n", p->x, p->y);
    //
    p->x = 3.4;
    p->y = 4.5;
}

void mainPoint() {
    point_t point = {1.2, 2.3};
    changePoint(&point);
    printf("x=%lf, y=%lf\n", point.x, point.y);
    //
}
```

x=1.2, y=2.3

x=3.4, y=4.5

# Const Struct Parameter

- What if you do not want the recipient to be able to modify the structure?
  - Use the const modifier

```
(const point_t * p)
```

# Using the `const` Modifier

```
typedef struct {
    double x; // x coordinate
    double y; // y coordinate
} point_t;

void changePoint(const point_t * p) {
    printf("x=%lf, y=%lf\n", p->x, p->y);
    p->x = 3.4;
    p->y = 4.5;
}

void mainPoint() {
    point_t point = {1.2, 2.3};
    changePoint(&point);
    printf("x=%lf, y=%lf\n", point.x, point.y);
}
```

ch08.c: In function `âchangePointâ`:

ch08.c:213:7: error: assignment of member `âxâ` in read-only object  
ch08.c:214:7: error: assignment of member `âyâ` in read-only object

# Return Structure

- Scalar values (*int, float, etc*) are efficiently returned in CPU registers
- Historically, the structure assignments and the return of structures was not supported in C
- But, the return of *pointers (addresses)*, including pointers to structures, has always been supported





# Return Structure Pointer to Local Variable

```
typedef struct {
    // unsigned char will hold 0-255
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} pixel_t;

pixel_t * getEmptyPixel() {
    // empty pixel = zeros
    pixel_t p = {0, 0, 0};

    // return pointer to empty pixel
    return &p;
}

pixel_t ePixel;
pixel_t * pixelPtr;

pixelPtr = getEmptyPixel();

// Immediately use return
ePixel = *pixelPtr;
```



ch08.c: In function âgetEmptyPixelâ:

ch08.c:293:7: warning: function returns address of local variable



# Return Structure Pointer to Local Variable

- Reason: function is returning a pointer to a variable that was allocated on the stack during execution of the function



- Such variables are subject to being wiped out by subsequent function calls



## Function Return Structure Values

- It is possible for a function to return a structure.
- This facility depends upon the structure assignment mechanisms which copies one complete structure to another.
  - Avoids the unsafe condition associated with returning a pointer, but
  - Incurs the possibly extreme penalty of copying a very large structure



# Function Return Structure Values

```
typedef struct {
    // unsigned char will hold 0-255
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} pixel_t;

pixel_t getEmptyPixel() {
    // empty pixel = zeros
    pixel_t p = {0, 0, 0};

    // return pointer to empty pixel
    return p;
}

pixel_t ePixel;

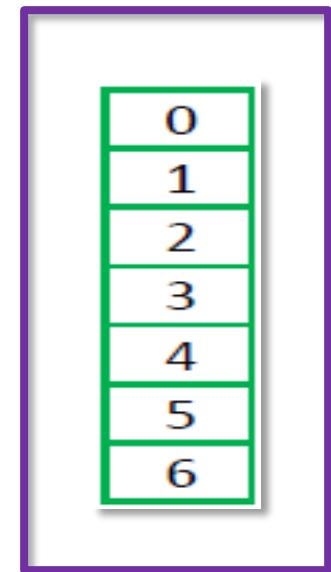
ePixel = getEmptyPixel();
```





# Arrays as Parameters & Return

- Array's address is passed as parameter
  - Simulates passing by reference
- Embedding array in structure
  - The only way to pass an array by value is to embed it in a structure
  - The only way to return an array is to embed it in a structure
  - Both involve copying
    - Beware of size



# Programming in C



## Chapter 9 Structures



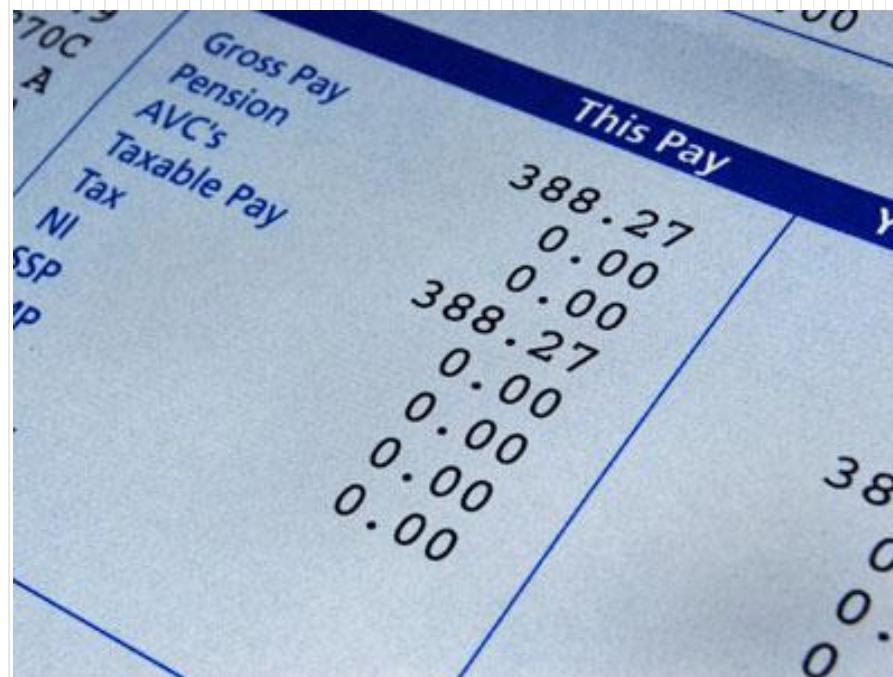
*THE END*

# Programming in C



# Chapter 8 App

## Payroll Lookup using Structure





2:frog26.cs.clemson.edu - default - SSH Secure Shell

File Edit View Window Help

Quick Connect Profiles

UW PICO 5.04 File: payroll.dat Modified

```
1111111111 40 10
2222222222 50 12.5
3333333333 45 8.5
4444444444 50 9
5555555555 30 7
6666666666 20 8.55
7777777777 40 12
8888888888 40 11.11
9999999999 45 15
```

Get He Write O Read F Y Prev P K Cut Te C Cur Po  
X Exit J Justif W Where V Next U UnCut T To Spe

Connected to frog26.cs.clemson.edu SSH2 - aes128-cbc - hmac-md5 - nc 56x20



# Memory Allocation & Structure

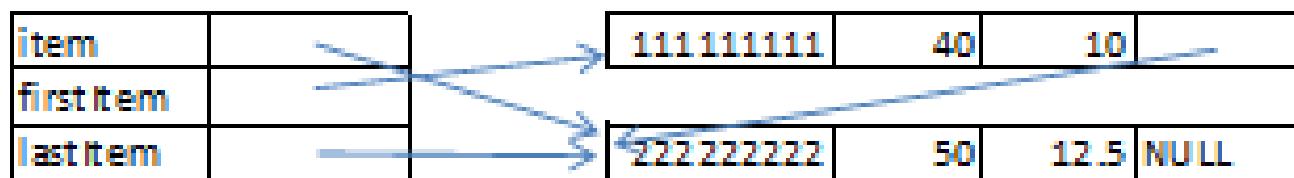
## Initialization

|           |      |
|-----------|------|
| item      | NULL |
| firstItem | NULL |
| lastItem  | NULL |

## First Record



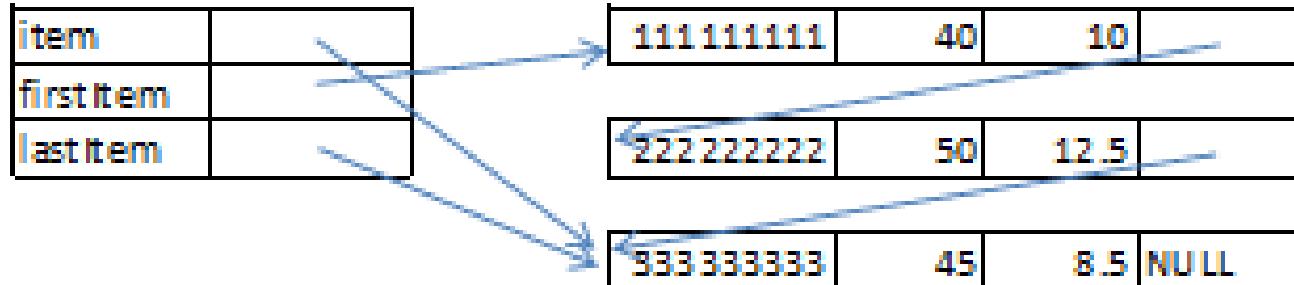
## Second Record



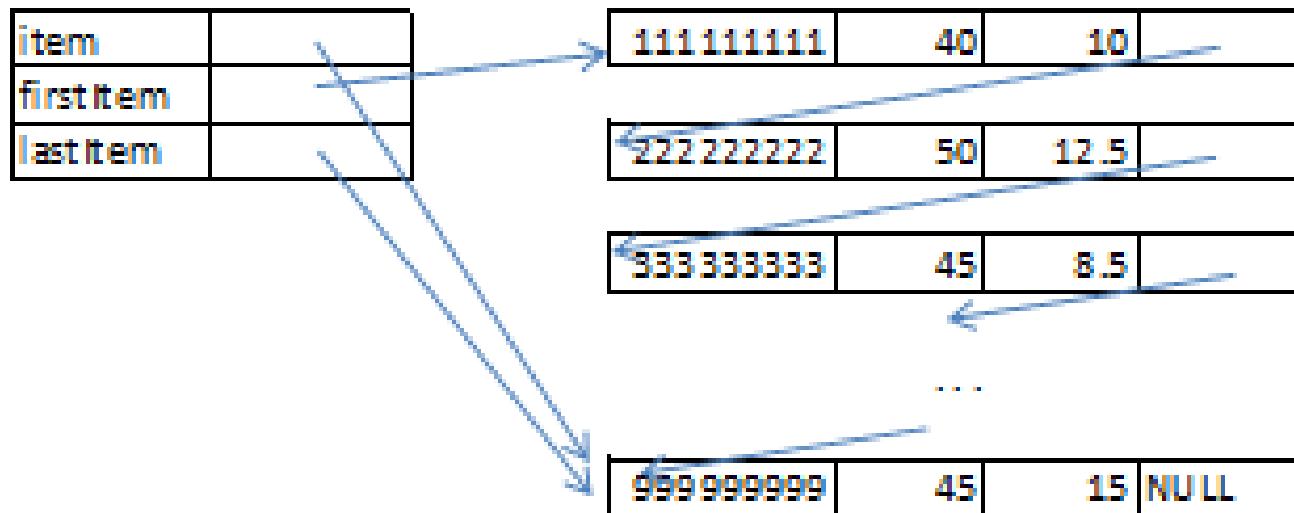


# Memory Allocation & Structure

Third Record



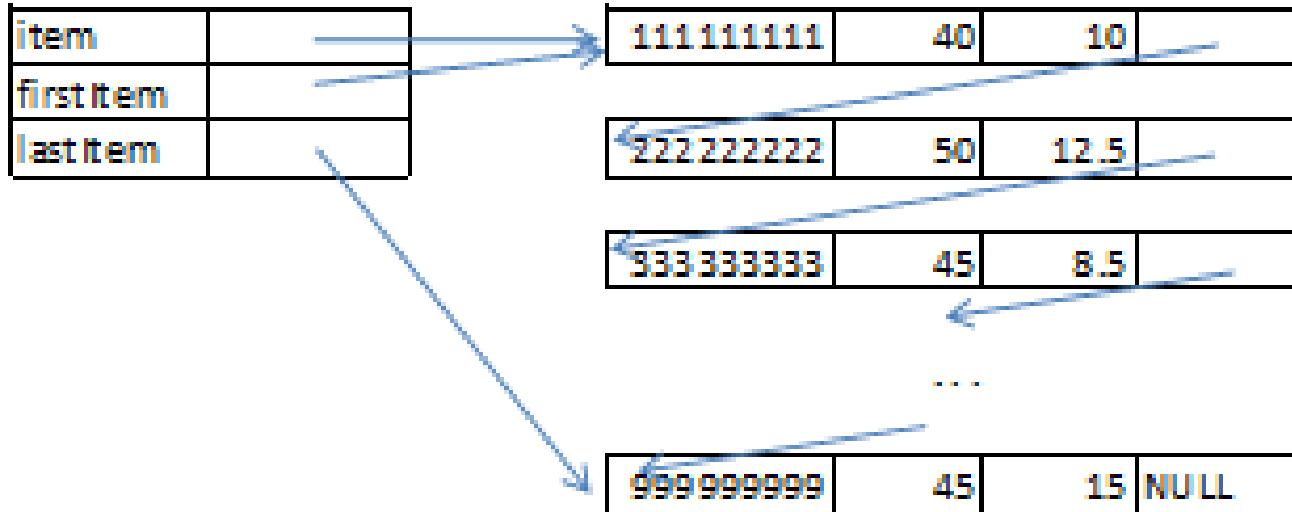
Last Record





# Traversing Records

Search Start





# ch08PayType.h

```
/*
Header:    ch08PayType
Purpose:   Definition of pay_t
Author:    Ima Programmer
Date:     mm/dd/yy
*/
struct pay_t {
    long id;
    int hrs;
    float rate;
    struct pay_t * nextPtr;
};
```

# ch08Pay.c



```
/*
Program: ch08Pay
Purpose: Lookup payroll record using struct
Author: Ima Programmer
Date: mm/dd/yy
*/
#include <stdio.h>
#include <stdlib.h>
#include "ch08PayType.h"

int main(int argc, char *argv[])
{
    // initialization
    struct pay_t * item = NULL;
    struct pay_t * firstItem = NULL;
    struct pay_t * lastItem = NULL;

    FILE * payFile = NULL;
    long id;
    float hrs, rate, pay;
```



```
// Open input
if (argc != 2) {
    printf("\nInvalid number of arguments\n\n");
    exit(1);
}
payFile = fopen(argv[1], "r");
if (payFile == NULL) {
    printf("\nCannot open %s\n\n", argv[1]);
    exit(2);
}
```



```
// load payroll data
while (fscanf(payFile, "%ld %f %f", &id, &hrs, &rate) == 3) {
    // get new pay item
    item = malloc(sizeof(struct pay_t));
    if (item == NULL) {
        printf("\nUnable to allocate memory for %ld!\n\n", id);
        exit(3);
    }
    if (firstItem == NULL)
        firstItem = item;
    else
        (*lastItem).nextPtr = item;
        // or lastItem->nextPtr = item
    lastItem = item;

    // load item data
    item->id = id;                // or (*item).id = id
    item->hrs = hrs;
    item->rate = rate;
    item->nextPtr = NULL;
} // load
fclose(payFile);
```



```
// lookup: process ids until zero
printf("\nEnter id or zero to end: ");
scanf("%ld", &id);
while (id != 0) {
    // lookup item
    item = firstItem;
    while (item != NULL && item->id != id)
        item = item->nextPtr;

    if (item == NULL) // not found
        printf("%ld not found\n", id);
    else { // found
        if ((*item).hrs <= 40)
            pay = item->hrs * item->rate;
        else
            pay = 40 * item->rate + (item->hrs - 40) * item->rate * 1.5;
        printf("Hours = %d, Rate = %f, Pay = %.2f\n",
               item->hrs, item->rate, pay);
    }
}

// next id
printf("\nEnter id or zero to end: ");
scanf("%ld", &id);
}
```



```
// free memory
while (firstItem != NULL) {
    item = firstItem;
    firstItem = item->nextPtr;
    free(item);
}

printf("\n");
return 0; // normal return
} // main
```



# Programming in C



Chapter 8 App  
Payroll Lookup using Struct

THE END

# Programming in C



## Chapter 9

### Strings

|    |          |          |          |           |          |           |
|----|----------|----------|----------|-----------|----------|-----------|
| s1 | 0        | 1        | 2        | 3         |          |           |
|    | <b>o</b> | <b>n</b> | <b>e</b> | <b>\0</b> |          |           |
| s2 | 0        | 1        | 2        | 3         |          |           |
|    | <b>t</b> | <b>w</b> | <b>o</b> | <b>\0</b> |          |           |
| s3 | 0        | 1        | 2        | 3         |          |           |
|    | <b>b</b> | <b>a</b> | <b>t</b> | <b>\0</b> |          |           |
| s4 | 0        | 1        | 2        | 3         | 4        | 5         |
|    | <b>h</b> | <b>e</b> | <b>l</b> | <b>l</b>  | <b>o</b> | <b>\0</b> |
| s5 | 0        | 1        | 2        | 3         |          |           |
|    | <b>b</b> | <b>y</b> | <b>e</b> | <b>\0</b> |          |           |

# Strings

- We've used strings

```
printf ("Hello");
```

“Hello” is string  
literal constant

- Array with base type char
  - One character per element
  - One extra character: '\0'
    - Called ‘null character’
    - End marker
  - Literal "Hello" stored as string



# String Variable Declaration

- Array of characters:

```
char s[10];
```

- Declares a c-string variable to hold up to 9 characters plus one null character
- No initial value

| s[0] | s[1] | s[2] | s[3] | s[4] | s[5] | s[6] | s[7] | s[8] | s[9] |
|------|------|------|------|------|------|------|------|------|------|
| ?    | ?    | ?    | ?    | ?    | ?    | ?    | ?    | ?    | ?    |

# String Variable

- Typically a partially filled array
  - Declare large enough to hold max-size string, including the null character.
- Given a standard array:

```
char s[10];
```
- If s contains string “Hi Mom！”, then stored as:

| s[0] | s[1] | s[2] | s[3] | s[4] | s[5] | s[6] | s[7] | s[8] | s[9] |
|------|------|------|------|------|------|------|------|------|------|
| H    | i    |      | M    | o    | m    | !    | \0   | ?    | ?    |





# String Variable Initialization

- Can initialize string:

```
char message[15] = "Hi There";
```

- Need not fill entire array
- Initialization places '\0' at end

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|
| H   | i   |     | T   | h   | e   | r   | e   | \0  | ?   | ?    | ?    | ?    | ?    | ?    |

# String Variable Initialization

- Can omit array-size:

```
char abc[] = "abc";
```

| [0] | [1] | [2] | [3] |
|-----|-----|-----|-----|
| a   | b   | c   | \0  |

- Automatically makes size one more than length of quoted string

- NOT same as:

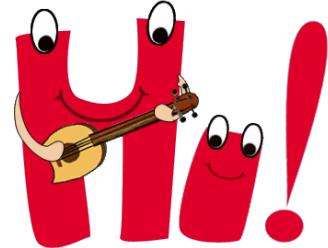
```
char abc[] = {'a', 'b', 'c'};
```

| [0] | [1] | [2] |
|-----|-----|-----|
| a   | b   | c   |

- IS same as:

```
char abc[] = {'a', 'b', 'c', '\0'};
```

| [0] | [1] | [2] | [3] |
|-----|-----|-----|-----|
| a   | b   | c   | \0  |



# String Indexes

- A string IS an array
- Can access indexed variables of:

```
char hi[5] = "Hi";
```

- hi[0] is 'H'
- hi[1] is 'i'
- hi[2] is '\0'
- hi[3] is unknown
- hi[4] is unknown

| [0] | [1] | [2] | [3] | [4] |
|-----|-----|-----|-----|-----|
| H   | i   | \0  | ?   | ?   |

# String Index Manipulation

- Can manipulate array elements

```
char dobedo[7] = "DoBeDe";
dobedo[5] = 'o';
dobedo[6] = '!';
```

| [0] | [1] | [2] | [3] | [4] | [5] | [6] |   |
|-----|-----|-----|-----|-----|-----|-----|---|
| D   | o   | B   | e   | D   | e   | \0  | ? |
| D   | o   | B   | e   | D   | o   | \0  | ? |
| D   | o   | B   | e   | D   | o   | !   | ? |

- Be careful!
- Here, '\0' (null) was overwritten by a '!'
- If null overwritten, string no longer 'acts' like a string!
  - Unpredictable results!



# String Library

- Used for string manipulations
  - Normally want to do ‘fun’ things with strings
  - Requires library string.h:

```
#include <string.h>
```



<http://en.wikipedia.org/wiki/String.h>

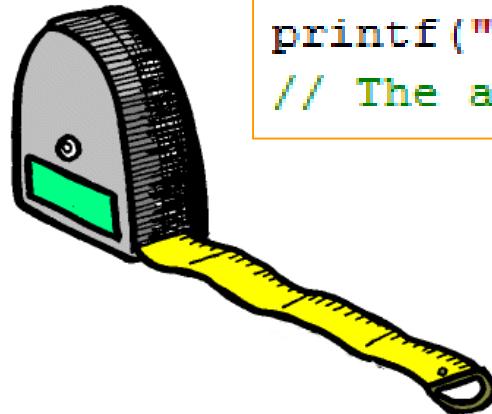
# String Length: strlen

- Often useful to know length of string

**strlen (string)**

- Returns number of characters
  - Does not include null
  - Return type is size\_t so type cast may be required

```
char hello_world[] = "Hello World";
printf("%d", (int) strlen(hello_world));
// The above will print number 11
```



# = with strings

- Strings are not like other variables, they are arrays

- Cannot assign:

```
char msg[10];  
msg = "Hello"; // ILLEGAL!
```

- Must use string library function for assignment:

**strcpy (destination, source)**



- NO checks for size – up to programmer!
  - ‘Assign’ value of msg to “Hello”:

```
strcpy(msg, "Hello");
```

- Or **strncpy (destination, source, limit)**

- No ending null character if limit is reached

# -- with strings

- Cannot use operator == to compare

```
char hello[] = "Hello";
char goodbye[] = "Goodbye";
if (hello == goodbye) // NOT ALLOWED
```

- Must use strcmp string library function to compare:

**strcmp(string1, string2)**

- Returns zero int if string1 is equal to string 2
- Returns <0 int if string1 is less than string2
- Returns >0 int if string1 is greater than string2

```
if (strcmp(hello, goodbye) == 0)
    printf("Hello equal to Goodbye");
else if (strcmp(hello, goodbye) < 0)
    printf("Hello less than Goodbye");
else
    printf("Hello greater than Goodbye");
```

# String Concatenate: strcat

- Appends one string onto end of another

**strcat(destination, source)**

```
char msg1[30] = "Hello";
char msg2[30] = "Hello";
strcat(msg1, "World"); // Result "HelloWorld"
strcat(msg2, " World"); // Result "Hello World"
```

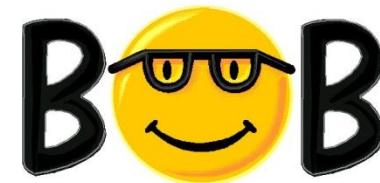
- Be careful when concatenating words
  - msg1 is missing space after Hello
  - msg2 is correct



# String Parameters to Functions

- A string is an array, so
  - String parameter is an array parameter
  - Strings passed to a function can be changed by the receiving function!
- Like all arrays, typical to send size as well
  - Function could also use '\0' to find end

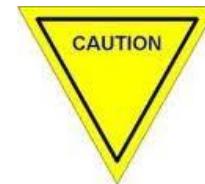
```
char msg[] = "BOB";
int msg_len = strlen(msg);
str_reverse(msg, msg_len);
```



# String Input and Output

- Watch input size of string
  - Must be large enough to hold entered string!
    - + '\n' perhaps
    - + '\0'
  - C gives no warnings of input size issues!

```
const int MAX_INPUT_STRING = 50;  
char input_string[MAX_INPUT_STRING + 2];
```



- Functions in stdio.h

# Character Input: getchar

- Reads one character at a time from a text stream

`int getchar( )`

- Reads the next character from the standard input stream and returns its value
- Return type is int!
  - Will convert if assigned to char

```
char in_ch;  
in_ch = getchar();
```

# Character Output: %s and putchar

- Format string placeholder for string: %s
- putchar: Writes one character at a time
  - Writes the parameter to standard output
  - If successful, returns the character written

```
char msg[] = "dlroW olleH";
int ndx;
// Print dlroW olleH
printf("%s\n", msg);
// Print Hello World
for (ndx = (int) strlen(msg) - 1; ndx >= 0; ndx--)
    putchar(msg[ndx]);
printf("\n");
```

String variable

# String Input: gets

`char *gets (char *strPtr)`

- Inputs a line (terminated by a newline) from standard input
- Converts newline to \0
- If successful, returns the string and also places it in argument
- Warning: Does not check length of input
  - gcc *may* produce warning message

```
char input_string[100];  
gets(input_string);
```

warning: the 'gets' function is dangerous and should not be used.



# String Input: fgets

`char *fgets (char * strPtr, int size, FILE *fp)`

- Inputs characters from the specified file pointer through \n or until specified size is reached
- Puts newline (\n) in the string if size not reached!!!
- Appends \0 at the end of the string
- If successful, returns the string & places in argument

```
const int MAX_LINE = 100;
char line_in[MAX_LINE + 2];
int line_len;
fgets(line_in, MAX_LINE, stdin);
// Check for \n
line_len = strlen(line_in);
if (line_in[line_len-1] == '\n')
    line_in[line_len-1] = '\0';
```

String variable

Use stdin for now

String variable or constant

# String Output: puts

**int puts (const char \*strPtr)**

- Takes a null-terminated string from memory and writes it to standard output
- Writes \n in place of \0

```
char hello[] = "Hello";
puts(hello);
printf("-----\n");
/*
    Prints:
    hello
    -----
*/
```

# String Output: fputs

```
int fputs (const char *strPtr, FILE *fp)
```

- Takes a null-terminated string from memory and writes it to the specified file pointer
- Drops \0
- Programmer's responsibility: Make sure the newline is present at the appropriate place(s)

```
char line_out[100] = "Hello!\n";
fputs(line_out, stdout);
```

String variable

Use stdout for now

# Programming in C



## Chapter 9

### Strings

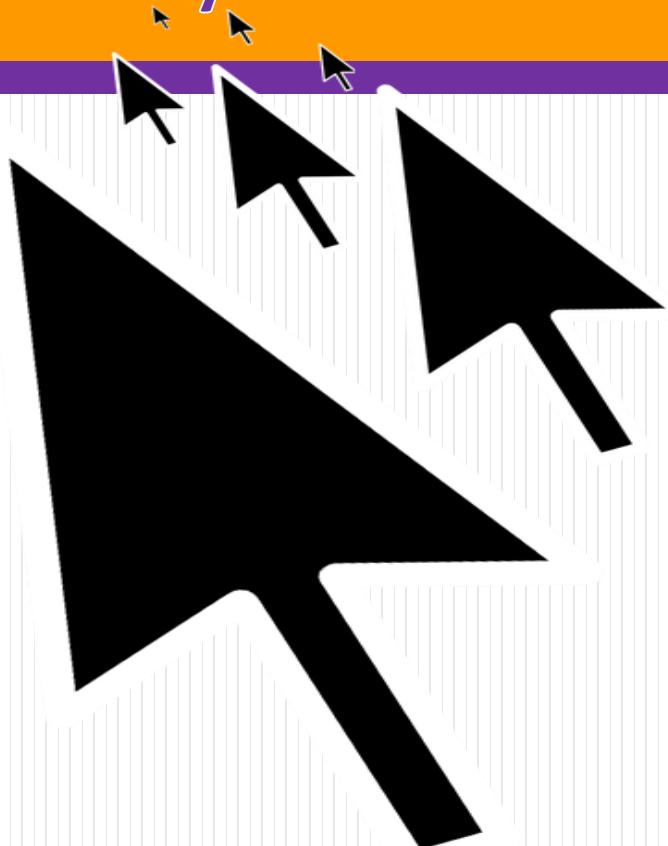
*THE END*

# Programming in C



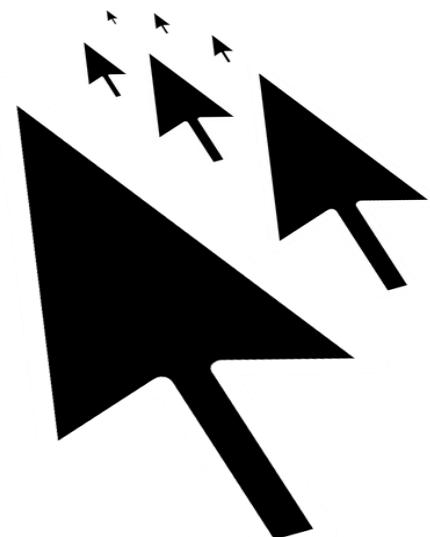
Chapter 10 / 16a (p.384)

Pointers / Dynamic Memory Allocation



# Pointer Variable

- A variable that stores a memory address
  - Allows C programs to simulate call-by-reference
  - Allows a programmer to create and manipulate dynamic data structures
- Must be defined before it can be used
  - Should be initialized to NULL or valid address



# Declaring Pointers

Declaration of pointers

`<type> *variable`

`<type> *variable = initial-value`

Examples:

```
int *x_ptr;      // Not initialized
double *aPtr = NULL, *bPtr = NULL;
char *grade = NULL;
```

# Pointers

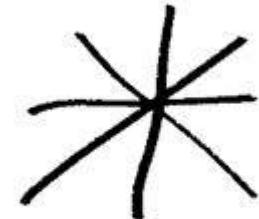
```
char *grade = NULL;
```

- A pointer variable has two associated values:
  - Direct value
    - address of another memory cell
    - Referenced by using the variable name
  - Indirect value
    - value of the memory cell whose address is the pointer's direct value.
    - Referenced by using the indirection operator \*



# Pointer Operators

- Come before a variable name
  - \* operator
    - Indirection operator or dereferencing operator
    - Returns a synonym, alias or nickname to which its operand points
  - & operator
    - Address of operator
    - Returns the address of its operand



# Pointer Variables

- One way to store a value in a pointer variable is to use the & operator

```
int count = 5;  
int *countPtr = &count;
```

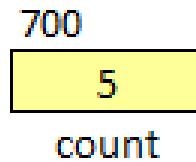
- The address of count is stored in countPtr
- We say, *countPtr points to count*

# Pointer Variables

- Assume count will be stored in memory at location 700 and countPtr will be stored at location 300

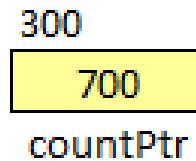
- `int count = 5;`

causes 5 to be stored in count



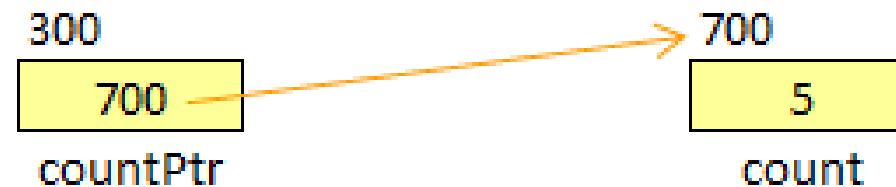
- `int *countPtr = &count;`

causes the address of count to be stored in countPtr



# Pointer Variables

We represent this graphically as



# Pointer Variables

- The indirection / dereferencing operator is \*
- `*countPtr = 10;`  
stores the value 10 in the address pointed to by countPtr



# Pointer Variables

- The character \* is used in two ways:

1. To declare that a variable is a pointer
  - Pre-pending a variable with a \* in a declaration declares that the variable will be a pointer to the indicated type instead of a regular variable of that type
2. To access the location pointed to by a pointer
  - Pre-pending a variable with a \* in an expression indicates the value in the location pointed to by the address stored in the variable



# Simulating By Reference

- Invoked function uses \* in formal parameters



```
void increment(int *n) {  
    *n += 1; // or (*n)++;  
}
```

- Invoking function uses & in actual parameters



```
int count = 0;  
increment(&count);  
printf("%d\n", count); // Prints 1
```

# Pointer Variables and Arrays

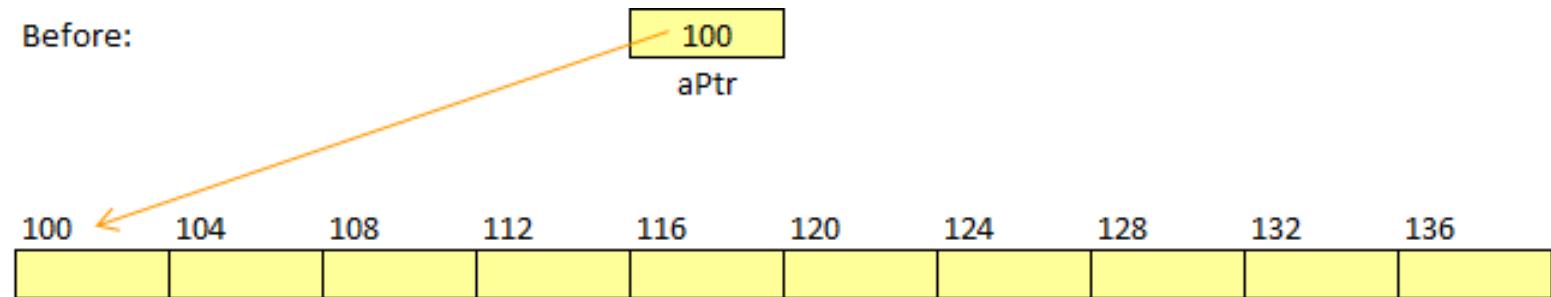
- Given

```
int x;  
int *xPtr = &x;  
*xPtr = 7;
```

- The compiler will know how many bytes to copy into the memory location pointed to by *xPtr*
- Defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code

# Pointer Variables and Arrays

- Consider a block in memory consisting of ten integers of 4 bytes in a row at location  $100_{10}$
- Now, let's say we point an integer pointer *aPtr* at the first of these integers

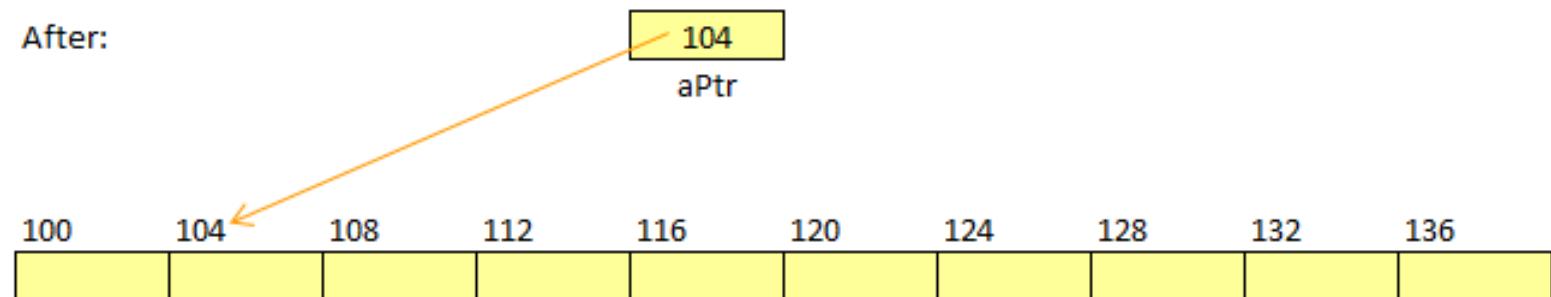


- What happens when we write

```
aPtr = aPtr + 1; ?
```

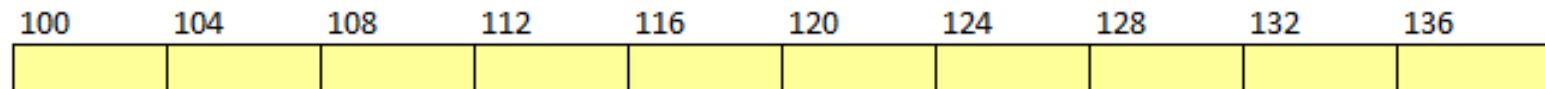
# Pointer Variables and Arrays

- Because the compiler "knows"
  - This is a pointer (i.e. its value is an address)
  - That it points to an integer of length 4 at location 100
- Instead of 1, `aPtr = aPtr + 1;` adds 4 to *aPtr*
  - Now *aPtr* "points to" the next integer at location 104
  - Same for: *aPtr+=1*, *aPtr++*, and *++aPtr*



# Pointer Variables and Arrays

- Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers

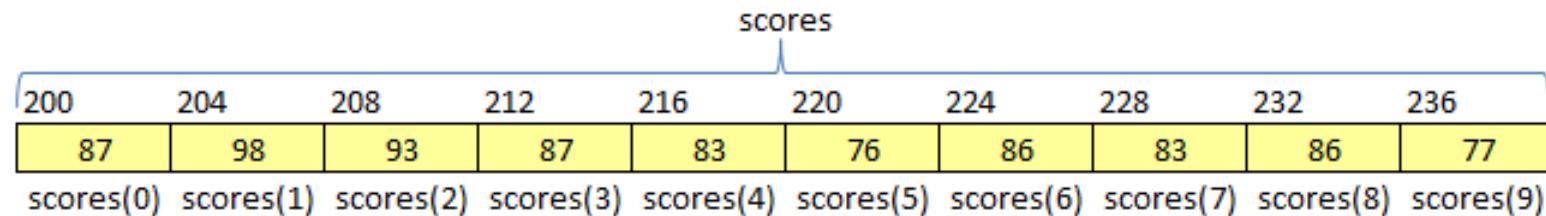


# Pointer Variables and Arrays

- Consider this array allocated at location 200

```
int scores[10] = {87, 98, 93, 87, 83, 76, 86, 83, 86, 77};
```

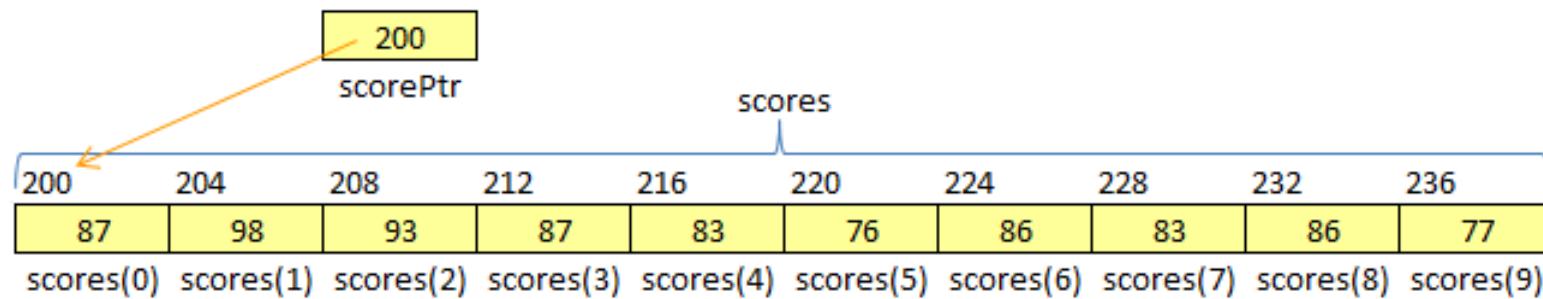
- We have an array containing 10 integers
- We refer to each of these integers by means of a subscript to *scores*
  - Using *scores[0]* through *scores[9]*



# Pointer Variables and Arrays

- The name of an array and the address of the first element in the array represent the same thing
- Consequently, we could alternatively access them via a pointer:

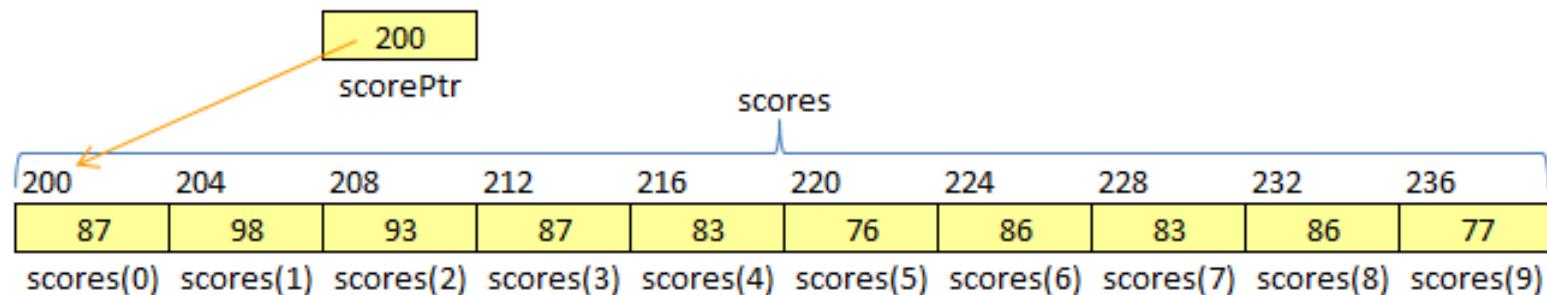
```
int scores[10] = {87, 98, 93, 87, 83, 76, 86, 83, 86, 77};  
...  
int *scorePtr = NULL;  
...  
scorePtr = &scores[0]; // Points to first element
```



# Pointer Variables and Arrays

- The name of an array is a pointer constant to the first element of the array
- So, we could also use :

```
int scores[10] = {87, 98, 93, 87, 83, 76, 86, 83, 86, 77};  
...  
int *scorePtr = NULL;  
...  
scorePtr = scores; // Points to array
```



# Pointer Arithmetic and Arrays

- If `scorePtr` is pointing to a specific element in the array and `n` is an integer,  
**`scorePtr + n`**  
is the pointer value  $n$  elements away
- We can access elements of the array either using the array notation or pointer notation
  - If `scorePtr` points to the first element, the following two expressions are equivalent:

**`scores[n]`**

**Array notation**

**`* (scorePtr + n)`**

**Pointer notation**

# Pointers and Dynamic Allocation of Memory



- So far, we have always allocated memory for variables that are located on the **stack**
  - Size of such variables must be known at compile time
- Sometimes convenient to allocate memory at run time
  - System maintains a second storage area called the **heap**
  - Functions **calloc** and **malloc** allocate memory as needed of size needed





# Pointers and Dynamic Allocation of Memory

1. Use allocating function (such as **malloc()**, **calloc()**, etc.)
  - Returns void pointer
    - void \* indicates a pointer to untyped memory
    - Will have to cast the returned value to the specific type needed
2. Use memory through the pointer notation
3. Release allocated space when no longer needed, so that it can be reused

# Pointers and Dynamic Allocation of Memory: `calloc`

- **`calloc`**
  - Used to dynamically create an array in the heap
  - Contiguous allocation
    - Initialized to binary zeros
  - Must `#include <stdlib.h>`
  - Takes two arguments
    1. Number of array elements
    2. Amount of memory required for one element
      - Use `sizeof` function / operator
  - Returns
    - Void pointer if successful
    - NULL if unsuccessful

# Pointers and Dynamic Allocation of Memory: calloc

## ■ Example 1: String

```
const int str_len = 500;
char *str_ptr = NULL;

...
→ str_ptr = (char *) calloc(str_len, sizeof(char));
if (str_ptr == NULL) {
    printf("Halting: Unable to allocate string.\n");
    exit(1);
}
```

## ■ Example 2: Integers

```
const int arraySize = 1000;
int *arrayPtr = NULL;

...
→ arrayPtr = (int *) calloc(arraySize, sizeof(int));
if (arrayPtr == NULL) {
    printf("Halting: Unable to allocate array.\n");
    exit(1);
}
```

# Pointers and Dynamic Allocation of Memory: malloc

## ■ malloc

- Used to dynamically get memory from heap
- Contiguous allocation
  - No initialization
- Must `#include <stdlib.h>`
- Takes one argument
  - Total amount of memory required
- Returns
  - Void pointer if successful
  - NULL if unsuccessful

# Pointers and Dynamic Allocation of Memory: malloc

## ■ Example 1: String

```
const int str_len = 500;
char *str_ptr = NULL;
...
→ str_ptr = (char *) malloc(str_len);
if (str_ptr == NULL) {
    printf("Halting: Unable to allocate string.\n");
    exit(1);
}
```

## ■ Example 2: Integers

```
const int arraySize = 1000;
int *arrayPtr = NULL;
...
→ arrayPtr = (int *) malloc(arraySize * sizeof(int));
if (arrayPtr == NULL) {
    printf("Halting: Unable to allocate array.\n");
    exit(1);
}
```

# Pointers and Dynamic Allocation of Memory: free

- **free**

- Used to dynamically release memory back to heap
- Contiguous deallocation
- Must `#include <stdlib.h>`
- Takes one argument
  - Pointer to beginning of allocated memory
- Good idea to also NULL pointer if reusing



# Pointers and Dynamic Allocation of Memory: free

## ■ Example 2 with free

```
const int arraySize = 1000;
int *arrayPtr = NULL;
...
arrayPtr = (int *) malloc(arraySize * sizeof(int));
if (arrayPtr == NULL) {
    printf("Halting: Unable to allocate array.\n");
    exit(1);
}
...
free(arrayPtr);
arrayPtr = NULL;
```



# Programming in C



Chapter 10 / 16a (p.384)

Pointers / Dynamic Memory Allocation

*THE END*

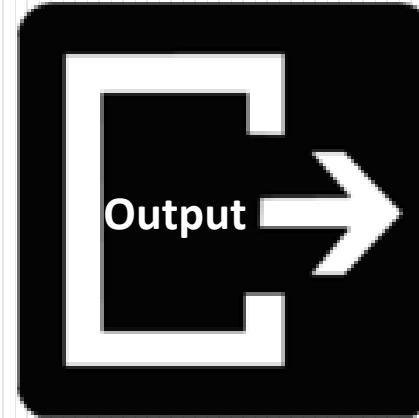
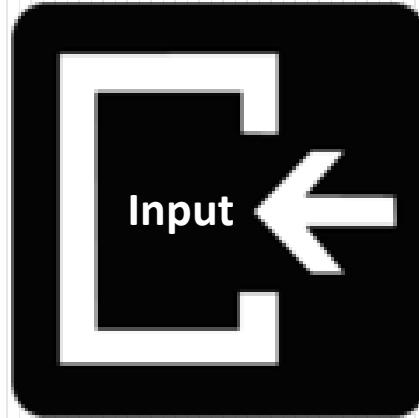


# Programming in C



## Chapter 15

### File Input/Output





# Standard File Pointers

- Assigned to console unless redirected
- Standard input = stdin
  - Used by scan function
  - Can be redirected: cmd < input-file
- Standard output = stdout
  - Used by printf function
  - Can be redirected: cmd > output-file
- Standard error = stderr
  - Can be specified in fputs function instead of stdout
  - Can be redirected: cmd 2> output-file

# Files



- A collection of related data treated as a unit
- Two types
  - Text
  - Binary
- Stored in secondary storage devices
- Buffer
  - Temporary storage area that holds data while they are being transferred to or from memory.



# Text Files

- Data is mainly stored as human-readable characters
- Each line of data ends with a newline character
  - ↵ = \n

|            |    |       |   |
|------------|----|-------|---|
| C666666666 | 20 | 8.55  | ↵ |
| A222222222 | 50 | 12.5  | ↵ |
| F333333333 | 45 | 8.5   | ↵ |
| B444444444 | 50 | 9     | ↵ |
| G555555555 | 30 | 6     | ↵ |
| E111111111 | 40 | 10    | ↵ |
| H777777777 | 40 | 12    | ↵ |
| D888888888 | 40 | 11.11 | ↵ |
| I999999999 | 45 | 15    | ↵ |

# User File Steps

```
#include <stdio.h>
```

1. Declare a file pointer variable
  - Program connection to external user file
2. Open the file
  - Creates a structure to store information needed for processing file and buffer area(s)
  - Makes file pointer connection to structure
3. Use functions for input and/or output
  - Handles movement of data between program and buffer and between buffer and external device
4. Close the file
  - Writes the buffer to file if necessary
  - Frees up memory associated with file

# 1. File Pointer Declaration

`FILE * variable-name-list;`

- Defines variables of type FILE\*, file pointer
- Pointer is undefined unless initialized
  - If not initialized to another value, initialize to NULL
- Examples:

```
FILE * scores_in = NULL;    // Input file
FILE * scores_out = NULL;   // Output file
```

- Following slides will use `fp` for file pointer

## 2. fopen

`FILE * fopen(char * filename, char * mode)`

- Parameters

- filename – string that supplies the name of the file as known to the external world
  - Default path is current directory

| mode | Meaning                                                                                                                                                                                |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| r    | <p>Open file for reading</p> <ul style="list-style-type: none"><li>• If file exists, the marker is positioned at beginning</li><li>• If file does not exist, error returned</li></ul>  |
| w    | <p>Open text file for writing</p> <ul style="list-style-type: none"><li>• If file exists, it is emptied</li><li>• If file does not exist, it is created</li></ul>                      |
| a    | <p>Open text file for append</p> <ul style="list-style-type: none"><li>• If file exists, the marker is positioned at the end</li><li>• If file does not exist, it is created</li></ul> |

# fopen

`FILE * fopen(char * filename, char * mode)`

- Return
  - If successful, file pointer
  - If not successful, NULL
  - Always check return
    - If not successful, print error message and exit or some other corrective action



# fopen

`FILE * fopen(char * filename, char * mode)`

- Examples

```
// Define and then open scores.txt for input
FILE * scores_in = NULL;
scores_in = fopen("scores.txt", "r");
if (scores_in == NULL) {
    printf("Unable to open scores.txt\n");
    exit(1);
}

// Define and open newscores.txt for output
FILE * scores_out = fopen ("newscores.txt", "w");
if (scores_out == NULL) {
    printf("Unable to open newscores.txt\n");
    exit(1);
}
```

## 4. fclose

**int `fclose(FILE *fp)`**

- Used to close a file when no longer needed
- Prevents associated file from being accessed again
- Guarantees that data stored in the stream buffer is written to the file
- Releases the FILE structure so that it can be used with another file
- Frees system resources, such as buffer space
- Returns zero on success, or EOF on failure

# fclose

- Examples:

```
fclose(scores_in);  
fclose(scores_out);
```

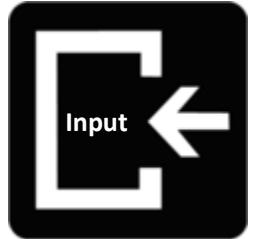


- To go back to beginning without fclose then fopen:  
**void rewind(FILE \*fp)**



# 3. Input/Output Functions

- Formatted Input
  - fscanf
- Formatted Output
  - fprintf
- String Input
  - fgets
- String Output
  - fputs



# Formatted Input Functions

- Read and convert a stream of characters and store the converted values in a list of variables found in the address list

- **scanf**

```
scanf("format string", address list);
```

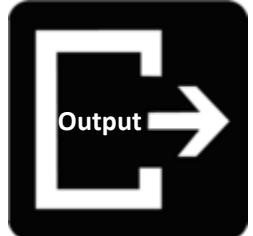
- Reads text data from standard input

- **fscanf**

```
fscanf(fp, "format string", address list);
```

- Reads input from the specified file

```
fscanf(scores_in, "%d", &score);
```



# Formatted Output Functions

- Displays output in human readable form
- `printf`

```
printf("format string", value list);
```

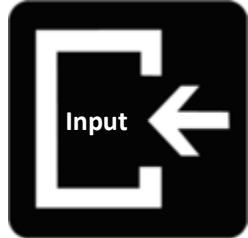
- Writes to standard output or standard error file

- `fprintf`

```
fprintf (fp, "format string", value list);
```

- Writes to the specified file

```
fprintf(scores_out, "%d\n", score);
```



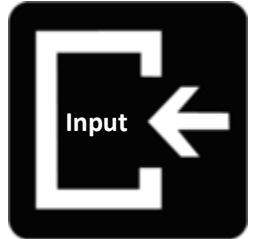
# String Input

- Reminder: Watch size of string
  - Must be large enough to hold largest input string
    - Plus \n perhaps
    - Plus \0 perhaps
  - C generally gives no warning of this issue

```
char input_string[MAX_INPUT_LENGTH+2];
```
- Standard Input
  - getchar: Read one character and return value as int

```
int getchar()
```
  - gets(): Read line & convert \n to \0, no size check

```
char *gets (char *strPtr)
```

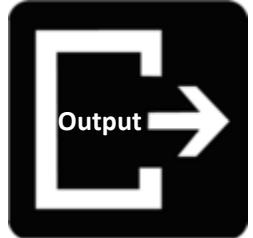


# String Input: fgets

```
char *fgets (char * strPtr, int size, FILE *fp)
```

- Inputs characters from the specified file pointer through \n or until specified size is reached
- Puts newline (\n) in the string if size not reached!!!
- Appends \0 at the end of the string
- If successful, returns the string & places in argument

```
const int MAX_LINE = 100;
char line_in[MAX_LINE + 2];
int line_len;
FILE * text_in = fopen("data.txt", "r");
// Should also check open return
fgets(line_in, MAX_LINE, text_in);
// Check for \n
line_len = strlen(line_in);
if (line_in[line_len-1] == '\n')
    line_in[line_len-1] = '\0';
```



# String Output

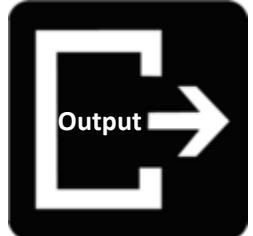
- Standard Output

- putchar: Write one character

```
int putchar(int outChar)
```

- puts(): Write line & converting \0 to \n

```
int puts (const char *strPtr)
```



# String Output: fputs

`int fputs (const char *strPtr, FILE *fp)`

- Takes a null-terminated string from memory and writes it to the specified file pointer
- Drops \0
- Programmer's responsibility: Make sure the newline is present at the appropriate place(s)

```
char line_out[100] = "Hello!\n";
FILE * msgFile = fopen("hello.txt", "w");
fputs(line_out, msgFile);
```

# End of File Controlled Loops

- `feof`

**`int feof(FILE *fp)`**

- Function to check if end of file has been reached.
- For an end of file controlled loop
  - Read before the loop
  - Test for end of file: `while (!feof(fp))`
  - Inside loop:
    - Process
    - Read at the bottom of the loop



# Programming in C



## Chapter 15

### File Input/Output

*THE END*

# Programming in C



## Chapter 16b

### Command Line Arguments

```
access.cs.clemson.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
[22:37:07] ~/cpsc111 [105] gcc -Wall ch17CmdArg.c
[22:37:14] ~/cpsc111 [106] ./a.out first second third fourth

5 arguments:
0: ./a.out
1: first
2: second
3: third
4: fourth

[22:37:24] ~/cpsc111 [107]
```

# Redirection



- Redirection: Read / Write to actual file
  - stdin: **cmd < input-file**
    - Ex: `./a.out < nums.txt`
  - stdout: **cmd > output-file**
    - Ex: `./a.out > report.txt`
  - stdout (append): **cmd >> output-file**
    - Ex: `./a.out >> report.txt`
  - Both: **cmd < input-file > output-file**
    - Ex: `./a.out < nums.txt > report.txt`

# Controlling the Command Line

- Command line arguments
  - It is often useful to pass arguments to a program via the command line. e.g.

```
gcc -Wall myProg.c -lm
```

- In this case, there are four command line arguments.
  - The count includes the command to execute the program.



# Command Line Arguments

- When a program is started from the command line,
  - The character strings (separated by spaces) comprising the program name and the remaining arguments are copied by the operating system into memory space occupied by the new program.
  - A table or array of addresses is passed to the main function. These values can be accessed by the main( ) function via arguments to main():

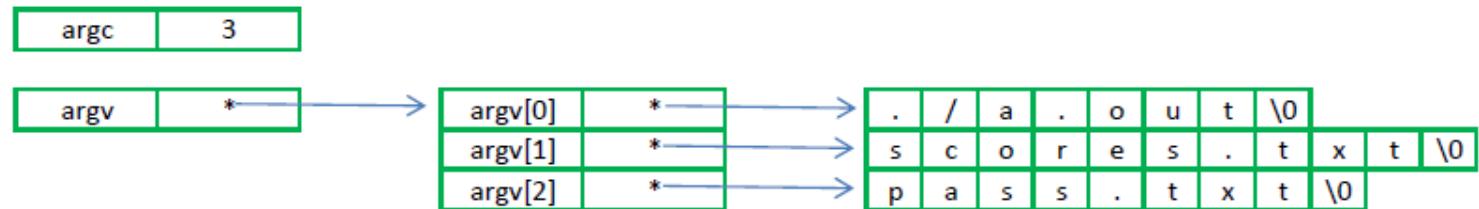
```
int main(int argc,      // count of number of command line arguments
         char *argv[]) // array of pointers to argument strings
```

- Parameter names argc and argv are used by convention but not required by C.

# Command Line Example

- **./a.out scores.txt pass.txt**

```
int main(int argc,      // count of number of command line arguments
         char *argv[]) // array of pointers to argument strings
```



# Printing Command Line Arguments

- Example:

```
int main(int argc, char *argv[])
{
    int index = 0;
    while (index < argc) {
        printf("%s\n", argv[index]);
        index++;
    }
}
```

# Printing Program Output

- When the program is invoked as follows:

```
./a.out input hello 5 mydata.dat
```

- Output is:

```
./a.out  
input  
hello  
5  
mydata.dat
```

# Add Command Line Numbers

```
int main(int argc, char *argv[]) {
    // variables
    int arg;
    float sum = 0;

    // verify command line arguments
    if (argc < 2) {
        printf("\nNumbers not specified on command line!\n\n");
        exit(1);
    }

    // add numbers
    for (arg = 1; arg < argc; arg++)
        sum += atof(argv[arg]);

    // print sum
    printf("\nSum is %f\n\n", sum);

    return 0;
}
```

atof = alpha (string) to float

./a.out 3.5 -1.1 4.725

Sum is 7.125000

# Programming in C



## Chapter 16b Command Line Arguments

*THE END*