CS 210 **October 20, 2016**
Day 14

Strings
Chapter 8

String Basics

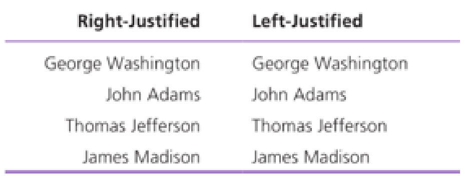
- A blank in a string is a valid character.
- null character
  - character '\0' that marks the end of a string in C
- A string in C is implemented as an array.
  - char string_var[30];
  - char str[20] = "Initial value";
- An array of strings is a 2-dimensional array of characters in which each row is a string.

Input/Output

- printf and scanf can handle string arguments
- use %s as the placeholder in the format string
- use a – (minus) sign to force left justification
  - printf("%-20s\n", president);

FIGURE 8.1

Right and Left Justification of Strings



Right-Justified	Left-Justified
George Washington	George Washington
John Adams	John Adams
Thomas Jefferson	Thomas Jefferson
James Madison	James Madison

```c
#include<stdio.h>
#pragma warning(disable:4996)
int main()
{
    char s1[] = "Hello Mom!";
    char s2[80];
    printf("%s\n", s1);
    //
    printf("Enter a string...");
    scanf_s("%s", s2, sizeof(s2)); //Requires buffer size
    printf("%s\n", s2);
    //
    printf("Enter a string...");
    scanf("%s", s2);   //Use pragma to enable
    printf("%s\n", s2);
}
```

Buffer Overflow

- more data is stored in an array than its declared size allows
- a very dangerous condition
- unlikely to be flagged as an error by either the compiler or the run-time system

String Assignment

- stcpy
  - copies the string that is its second argument into its first argument
    - strcpy(s1, “hello”);
  - subject to buffer overflow
- strncpy
  - take an argument specifying the number of characters to copy
  - if the string to be copies is shorter, the remaing characters stored are null
    - strncpy(s2, “inevitable”, 5);

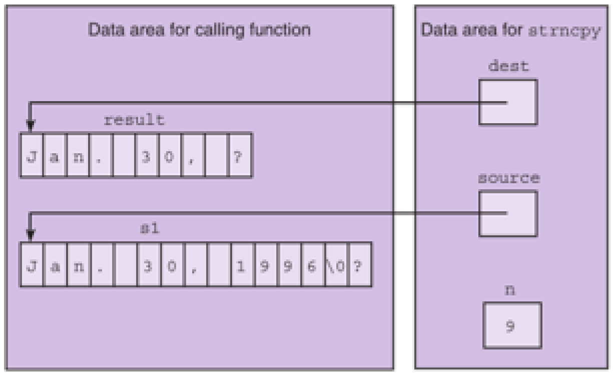
```c
#include<stdio.h>
#include<string.h>
#pragma warning(disable:4996)

int main()
{
    char s1[] = "Hello Mom!";
    char s2[80];
    char s3[80];
    strcpy_s(s2, sizeof(s1), s1);
    printf("%s\n", s2);
    //
    strcpy(s2, s1);    //Use pragma for this
    printf("%s\n", s2);
    //
    strncpy(s3, s1, 3);
    s3[3] = '\0';
    printf("%s\n", s3);
    //
    strncpy(s3, &s1[2], 6);
    s3[6] = '\0';
    printf("%s\n", s3);
    //
    //              01234567890123456789012
    char name[] = "Matilda M. McGillicuddy";
    char first[20], middle[3], last[20];
    int n = sizeof(name);
    strncpy(last, &name[11], 12);
    strncpy(middle, &name[8], 2);
    strncpy(first, name, 7);
    last[12] = '\0';
    printf("%s, ", last);
    first[7] = '\0';
    printf("%s, ", first);
    middle[2] = '\0';
    printf("%s\n", middle);

}
```

Substrings

- a fragment of a longer string

FIGURE 8.5

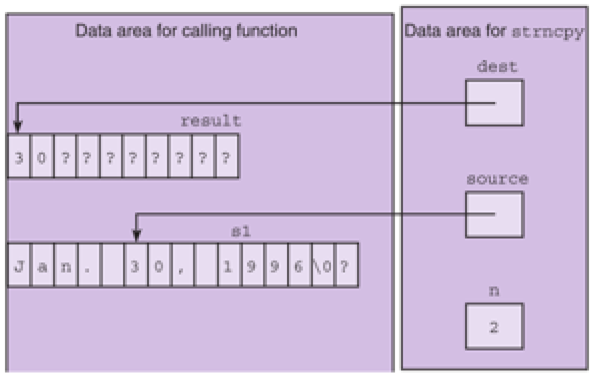
Execution of `strncpy(result, s1, 9);`


Data area for calling function
Data area for strncpy
dest
result
J a n .   3 0 ,   ?
source
s1
J a n .   3 0 ,   1 9 9 6 \0 ?
n
9

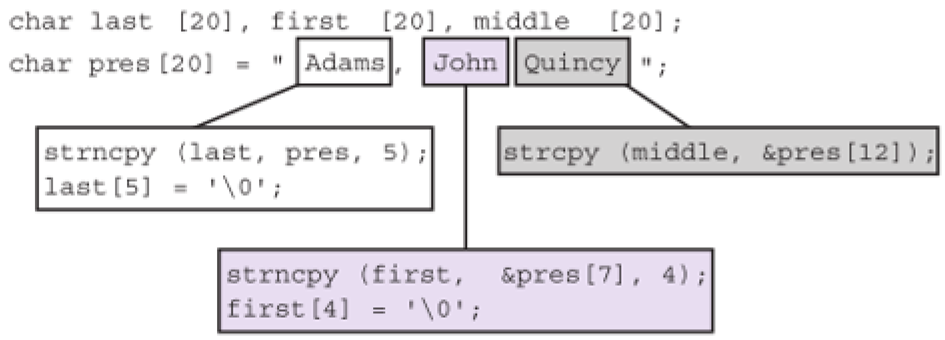
Substrings

FIGURE 8.6

Execution of `strncpy(result, &s1[5], 2);`


Data area for calling function
Data area for strncpy
dest
result
3 0 ? ? ? ? ? ? ? ?
source
s1
J a n .   3 0 ,   1 9 9 6 \0 ?
n
2

Substrings



```
char last [20], first [20], middle [20];
char pres[20] = " Adams, John Quincy ";
```

```
strncpy (last, pres, 5);
last[5] = '\0';
```

```
strcpy (middle, &pres[12]);
```

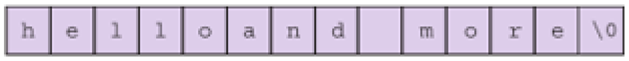
```
strncpy (first, &pres[7], 4);
first[4] = '\0';
```

String Terminology

- string length
  - in a character array, the number of characters before the first null character
- empty string
  - a string of length zero
  - the first character of the string is the null character

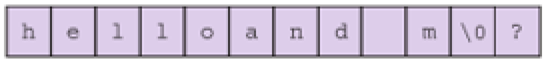
Concatenation

- strcat
  - appends source to the end of dest
  - assumes that sufficient space is allocated for the first argument to allow addition of the extra characters
    - s1 = “hello”;
    - strcat(s1, “and more”);



Concatenation

- strncat
  - appends up to n characters of source to the end of dest, adding the null character if necessary
  - assumes that sufficient space is allocated for the first argument to allow addition of the extra characters
    - s1 = “hello”;
    - strncat(s1, “and more”, 5);



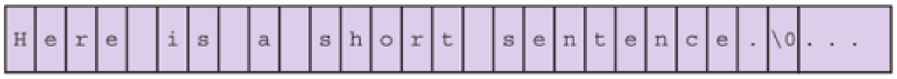
Scanning a Full Line

- For interactive input of one complete line of data, use the gets function.
- The \n character representing the <return> or <enter> key pressed at the end of the line is not stored.

Scanning a Full Line

```
char line[80];
printf("Type in a line of data.\n> ");
gets(line);
```

```
Type in a line of data.
> Here is a short sentence.
```



```
#include<stdio.h>
#include<string.h>
int main()
{
    char line[80];
    int i, cnt = 0;
    printf("Enter a line of data... \n");
    gets(line);
    printf("%s\n", line);
    for(i=0;i<sizeof(line);i++)
       if(line[i] == ' ')
          cnt++;
    printf("There are %d spaces in the line.\n", cnt);
}
```

CS 210 **October 20, 2016**
Strings

Write a program which inputs a single line from the user as a string. Calculate and print the number of lower case letters a to z and print this number to the console.

Turn in a printed copy of your source file.

A sample output might look like this:
```
Enter a line of data...
aaabbbccczzz
a = 3
b = 3
c = 3
d = 0
e = 0
f = 0
g = 0
h = 0
i = 0
j = 0
k = 0
l = 0
m = 0
n = 0
o = 0
p = 0
q = 0
r = 0
s = 0
t = 0
u = 0
v = 0
w = 0
x = 0
y = 0
z = 3
Press any key to continue . . .
```

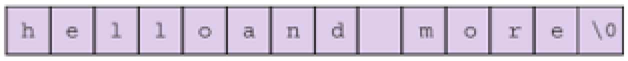
String Terminology

- string length
  - in a character array, the number of characters before the first null character

- empty string
  - a string of length zero
  - the first character of the string is the null character

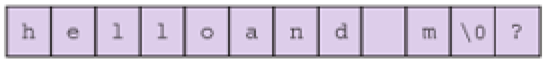
Concatenation

- strcat
  - – appends source to the end of dest
  - – assumes that sufficient space is allocated for the first argument to allow addition of the extra characters
    - s1 = “hello”;
    - strcat(s1, “and more”);



Concatenation

- strncat
  - – appends up to n characters of source to the end of dest, adding the null character if necessary
  - – assumes that sufficient space is allocated for the first argument to allow addition of the extra characters
    - s1 = “hello”;
    - strncat(s1, “and more”, 5);



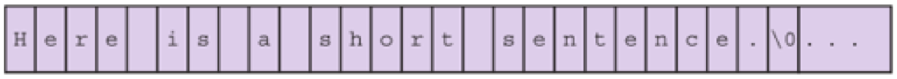
Scanning a Full Line

- For interactive input of one complete line of data, use the gets function.
- The \n character representing the <return> or <enter> key pressed at the end of the line is not stored.

Scanning a Full Line

```
char line[80];
printf("Type in a line of data.\n> ");
gets(line);
```

```
Type in a line of data.
> Here is a short sentence.
```



```
#include<stdio.h>
#include<string.h>
int main()
{
    char line[80];
    int i, cnt = 0;
    printf("Enter a line of data... \n");
    gets(line);
    printf("%s\n", line);
    for(i=0;i<sizeof(line);i++)
       if(line[i] == ' ')
          cnt++;
    printf("There are %d spaces in the line.\n", cnt);
}
```

There is also a version of gets for files called `fgets`. The following reads a text file called MyText.txt and prints it to the console using fgets.

```c
#define LINELEN 80
#include<stdio.h>
int main()
{
    char line[LINELEN];
    FILE *inp;  //Declare pointers to in
    int err;
    char *status;
    //Open the file for input
    err = fopen_s(&inp, "MyText.txt", "r");
    status = fgets(line, LINELEN, inp);
     while(status != 0)
     {
        printf("%s", line);
        status = fgets(line, LINELEN, inp);
     }
    fclose(inp);
}
```

String Comparison

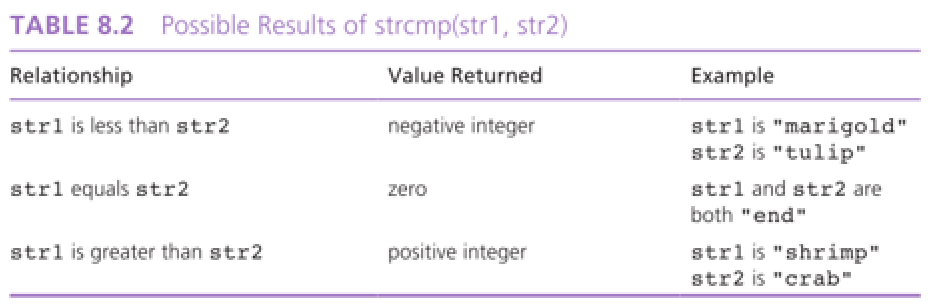
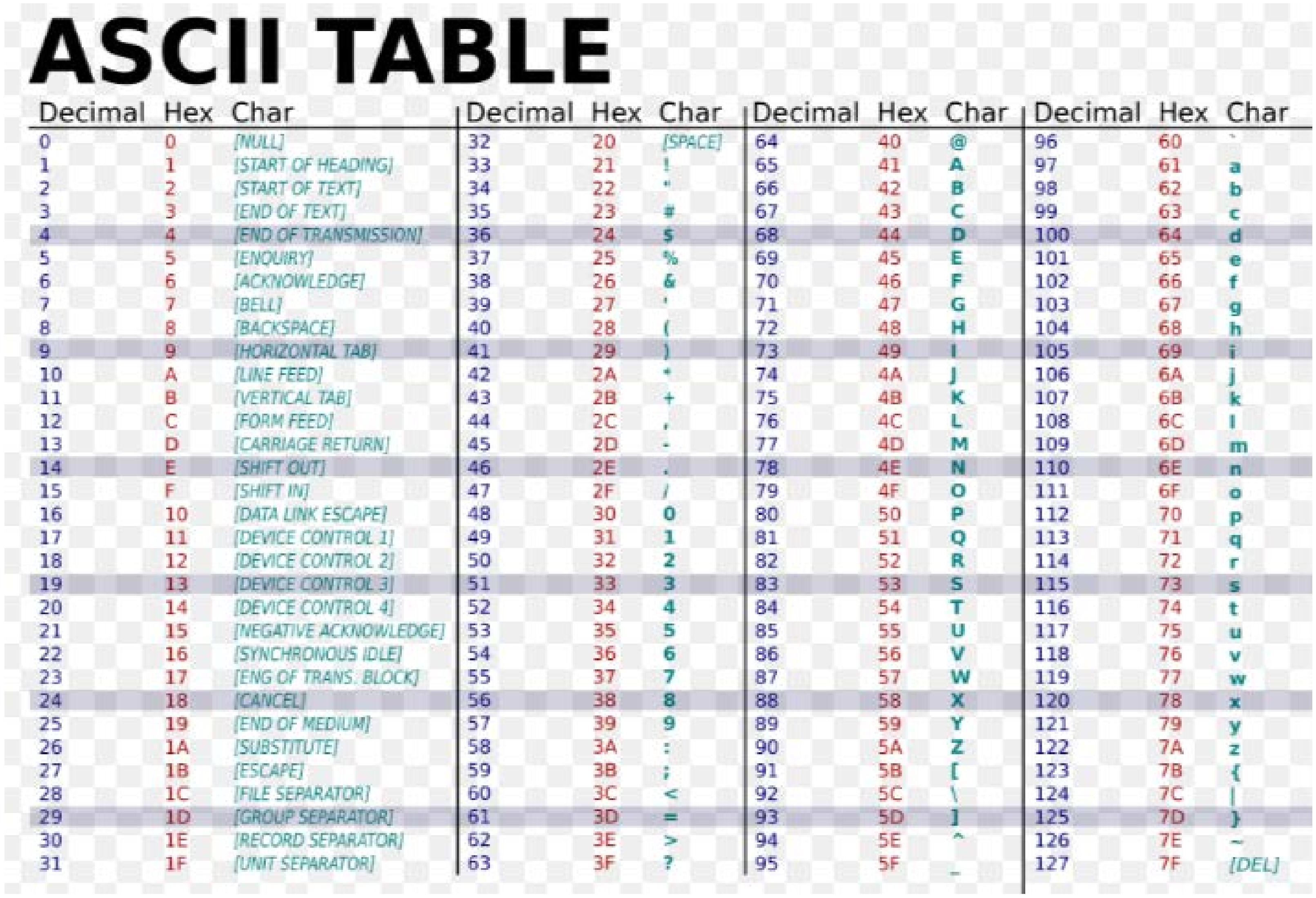


TABLE 8.2 Possible Results of strcmp(str1, str2)

Relationship	Value Returned	Example
`str1` is less than `str2`	negative integer	`str1` is `"marigold"` `str2` is `"tulip"`
`str1` equals `str2`	zero	`str1` and `str2` are both `"end"`
`str1` is greater than `str2`	positive integer	`str1` is `"shrimp"` `str2` is `"crab"`



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s1[] = "abc";
    char s2[] = "bcd";
    char s3[] = "abcd";
    char s4[] = "ABC";
    char s5[] = "123";
    printf("%s to %s = %d\n", s1, s2, strcmp(s1, s2));
    printf("%s to %s = %d\n", s1, s3, strcmp(s1, s3));
    printf("%s to %s = %d\n", s1, s4, strcmp(s1, s4));
    printf("%s to %s = %d\n", s1, s1, strcmp(s1, s1));
    printf("%s to %s = %d\n", s1, s5, strcmp(s1, s5));

}
/*
abc to bcd = -1
abc to abcd = -1
abc to ABC = 1
abc to abc = 0
abc to 123 = 1
Press any key to continue . . .
*/
```

Since strings are char arrays and string names are pointers to those arrays, if we have an array of strings we have an array of pointers where each pointer points to a string.

Suppose we have five color names and we put them in an array which is necessarily two-dimensional.

```
char color[5][10] = {"red", "orange", "yellow", "green", "blue"};
```

In memory this looks like this:

Address		0	1	2	3	4	5	6	7	8	9
100	0	r	e	d							
110	1	o	r	a	n	g	e				
120	2	y	e	l	l	o	w				
130	3	g	r	e	e	n					
140	5	b	l	u	e						

The addresses are the pointer values of each string: color[0] = 100, color[1] = 110, etc.

If we want to sort this array of colors into alphabetical order we can do it the traditional way using strcpy_s to move the strings around. The following program uses a Bubble sort

```
/*This program sorts an array of strings by sorting
    in the traditional manner using strcpy_s to
    swap strings in a Bubble sort
*/
void BubbleSort(char color[][STRSIZE]);
int main()
{
   char color[STRNUM][STRSIZE] = {"red", "orange", "yellow", "green", "blue"};
   int i;
   for(i=0;i<STRNUM;i++)
      printf("%s\n", color[i]);
   BubbleSort(color);
   printf("Sorted\n");
   for(i=0;i<STRNUM;i++)
      printf("%s\n", color[i]);
}
void BubbleSort(char color[][STRSIZE])
{
    int i, fDone;
    char tmp[STRSIZE];
    fDone = 0;
    while(!fDone)
        {fDone = 1;
         for(i=0;i<STRNUM-1;i++)
             {if(strcmp(color[i],color[i+1]) == 1)
                 {strcpy_s(tmp, STRSIZE, color[i]);
                  strcpy_s(color[i], STRSIZE, color[i+1]);
                  strcpy_s(color[i+1], STRSIZE, tmp);
                  fDone = 0;
                 }
             }
         }
     }
```

The lines:

```
              if(strcmp(color[i],color[i+1]) == 1)
                 {strcpy_s(tmp, STRSIZE, color[i]);
                  strcpy_s(color[i], STRSIZE, color[i+1]);
                  strcpy_s(color[i+1], STRSIZE, tmp);
                  fDone = 0;
                 }
```

do the swap of two strings using `strcpy_s` when two adjacent strings are out of order.

If we run this program the string array will be rearranged to look like this:

Address		0	1	2	3	4	5	6	7	8	9
100	0	b	l	u	e						
110	1	g	r	e	e	n					
120	2	o	r	a	n	g	e				
130	3	r	e	d							
140	5	y	e	l	l	o	w				

Note that in each case the entire color string has been moved to a new address. The color "blue" was at 140 and is now at 100 etc.

Alternatively, we need not move the strings. We could just move the pointers to the strings – that is, create a set of pointers and rearrange the pointers so that they are in order.

We can create an array of pointers and put these addresses (pointers) into the array like this:

```
char color[5][10] = {"red", "orange", "yellow", "green", "blue"};
char *colorPtr[5];
int i;
for(i=0;i<5;i++)
   colorPtr[i] = color[i];
```

In memory we have:

colorPtr[]

Index	Address	Value	Points to
0	550	100	-> red
1	551	110	-> orange
2	552	120	->yellow
3	553	130	->green
4	554	140	->blue

We can rearrange the pointers so that if the pointers are printed by index they point to the colors in the right order. Here is the program.

```
/*This program sorts an array of strings by sorting
    the pointers to those strings. It uses a bubble
    sort.
*/
void BubbleSort(char *cPtr[]);
int main()
{
   char color[STRNUM][STRSIZE] = {"red", "orange", "yellow", "green", "blue"};
   char *colorPtr[STRNUM];
   int i;
   for(i=0;i<STRNUM;i++)
      colorPtr[i] = color[i];
   for(i=0;i<STRNUM;i++)
      printf("%d => %s\n", colorPtr[i], colorPtr[i]);
   BubbleSort(colorPtr);
   printf("Sorted\n");
   for(i=0;i<STRNUM;i++)
      printf("%d => %s\n", colorPtr[i], colorPtr[i]);
}
void BubbleSort(char *cPtr[])
{
    int i, fDone;
    char *tmp;
    fDone = 0;
    while(!fDone)
        {fDone = 1;
         for(i=0;i<STRNUM-1;i++)
             {if(strcmp(cPtr[i],cPtr[i+1]) == 1)
                 {tmp = cPtr[i];
                  cPtr[i] = cPtr[i+1];
                  cPtr[i+1] = tmp;
                  fDone = 0;
                 }
             }
         }
     }
```

Note that the swap given by:

```
{if(strcmp(cPtr[i],cPtr[i+1]) == 1)
     {tmp = cPtr[i];
      cPtr[i] = cPtr[i+1];
      cPtr[i+1] = tmp;
      fDone = 0;
     }
```

is swapping pointers – not strings. After this program runs we have:

In memory we have:
`colorPtr[]`

Index	Address	Value	Points to
0	550	140	->blue
1	551	130	->green
2	552	110	->orange
3	553	120	->yellow
4	554	100	->red

Now when we print the data by pointer index it is sorted but the data has not moved – only the pointers have moved in the pointer array.

Character Input/Output

- getchar
  - get the next character from the standard input source (that scanf uses)
  - does not expect the calling module to pass the address of a variable to store the input character
  - takes no arguments, returns the character as its result

```
ch = getchar()
```

Character Input/Output

- getc
  - used to get a single character from a file
  - comparable to getchar except that the character returned is obtained from the file accessed by a file pointer (ex., inp)

```
getc(inp)
```

Character Input/Output

- putchar
  - single-character output
  - first argument is a type int character code
  - recall that type char can always be converted to type in with no loss of information

```
putchar('a');
```

Character Input/Output

- putc
  - identical to putchar except it sends the single character/int to a file, ex., outp

putc('a', outp);

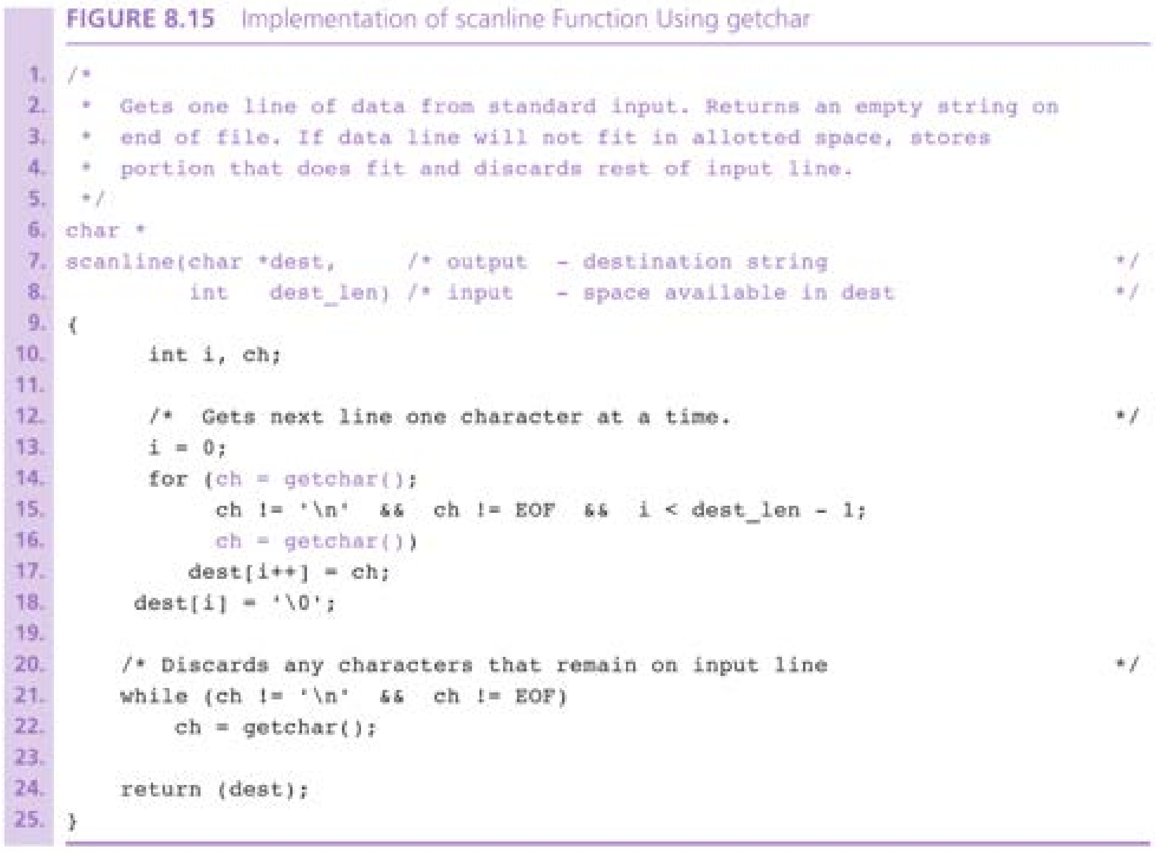


FIGURE 8.15 Implementation of scanline Function Using getchar

```
/*
 *  Gets one line of data from standard input. Returns an empty string on
 *  end of file. If data line will not fit in allotted space, stores
 *  portion that does fit and discards rest of input line.
 */
char *
scanline(char *dest,     /* output  - destination string                    */
         int   dest_len) /* input   - space available in dest               */
{
      int i, ch;

      /*  Gets next line one character at a time.                           */
      i = 0;
      for (ch = getchar();
           ch != '\n'  &&  ch != EOF  &&  i < dest_len - 1;
           ch = getchar())
         dest[i++] = ch;
     dest[i] = '\0';

    /* Discards any characters that remain on input line                    */
    while (ch != '\n'  &&  ch != EOF)
        ch = getchar();

    return (dest);
}
```

These character functions need `#include<ctype.h>`

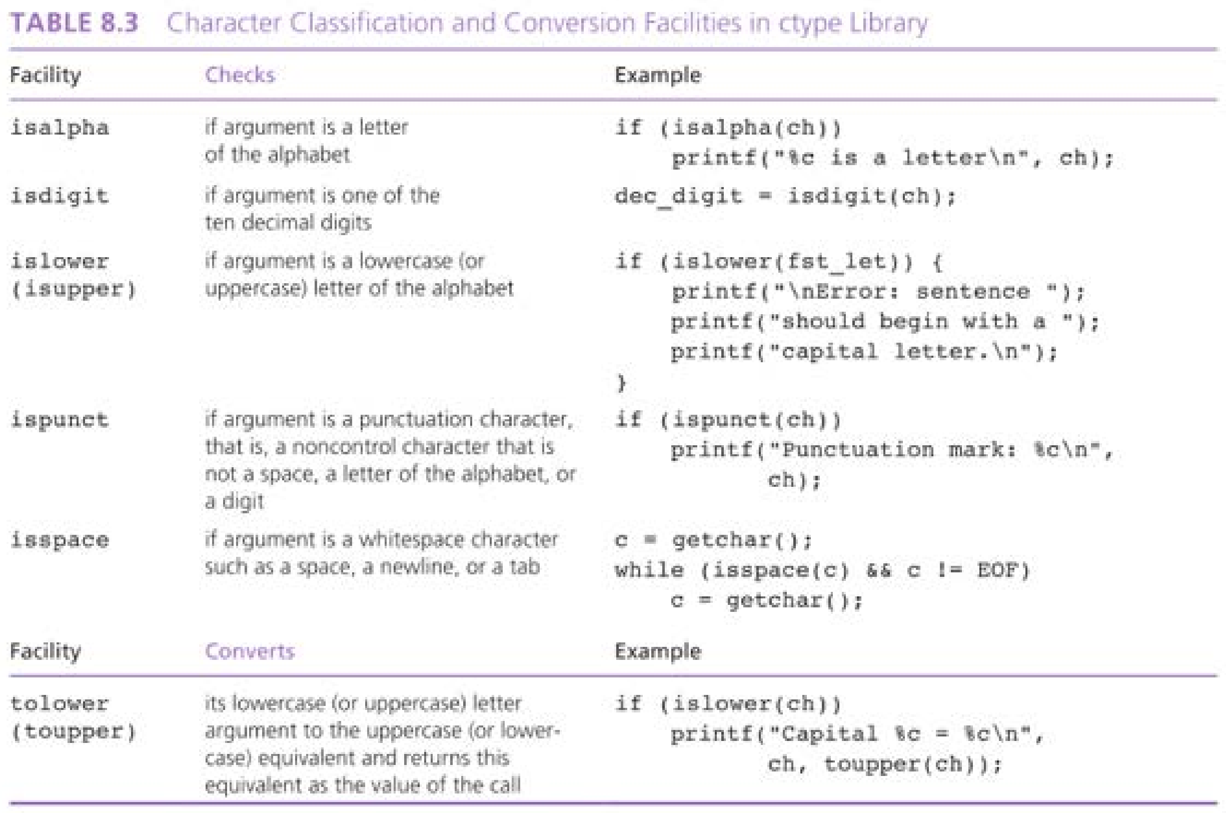


TABLE 8.3 Character Classification and Conversion Facilities in ctype Library

Facility	Checks	Example

`isalpha`	if argument is a letter of the alphabet	`if (isalpha(ch))`
		`    printf("%c is a letter\n", ch);`

`isdigit`	if argument is one of the ten decimal digits	`dec_digit = isdigit(ch);`

`islower` (`isupper`)	if argument is a lowercase (or uppercase) letter of the alphabet	`if (islower(fst_let)) {`
		`    printf("\nError: sentence ");`
		`    printf("should begin with a ");`
		`    printf("capital letter.\n");`
		`}`

`ispunct`	if argument is a punctuation character, that is, a noncontrol character that is not a space, a letter of the alphabet, or a digit	`if (ispunct(ch))`
		`    printf("Punctuation mark: %c\n",`
		`           ch);`

`isspace`	if argument is a whitespace character such as a space, a newline, or a tab	`c = getchar();`
		`while (isspace(c) && c != EOF)`
		`    c = getchar();`

Facility	**Converts**	**Example**

`tolower` (`toupper`)	its lowercase (or uppercase) letter argument to the uppercase (or lowercase) equivalent and returns this equivalent as the value of the call	`if (islower(ch))`
		`    printf("Capital %c = %c\n",`
		`           ch, toupper(ch));`

These string functions need `#include<string.h>`

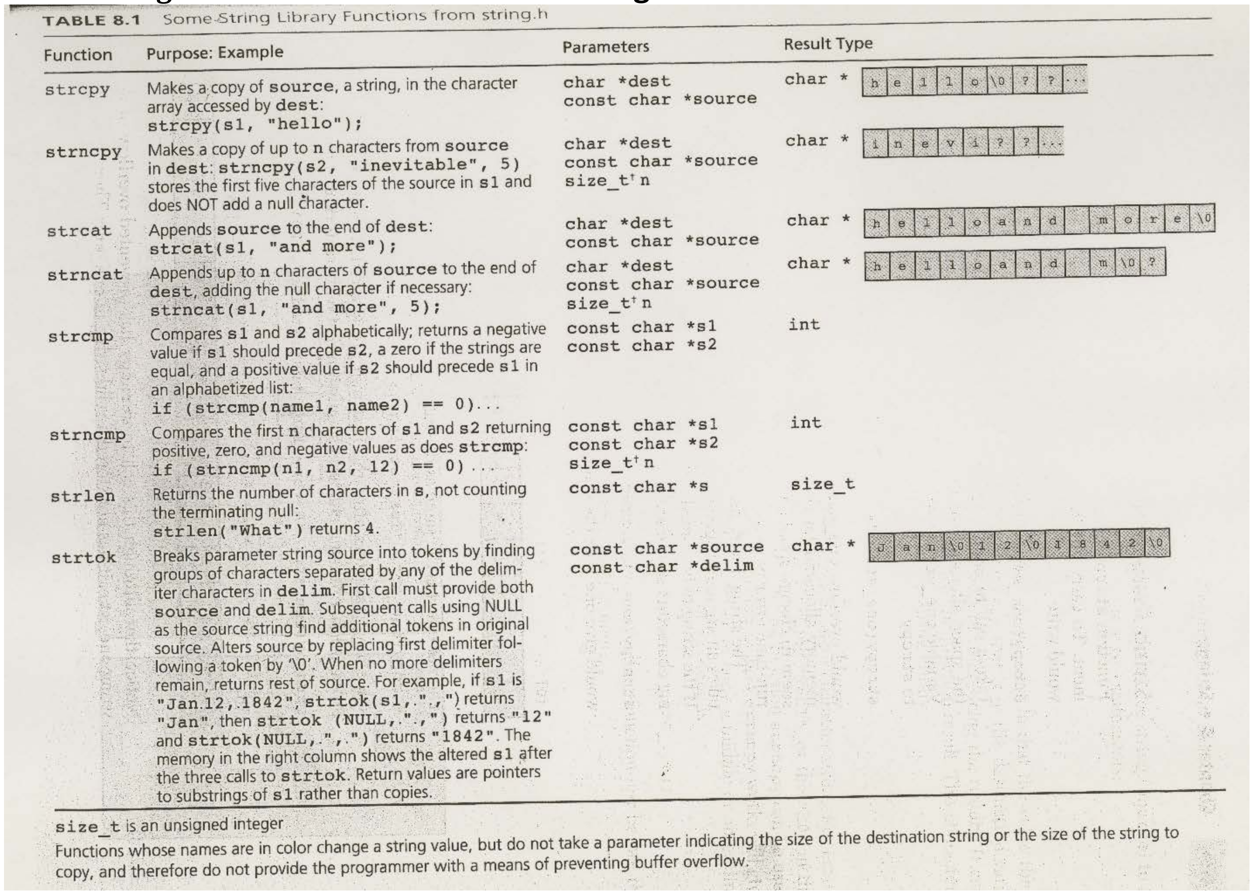


TABLE 8.1 Some String Library Functions from string.h

Function	Purpose: Example	Parameters	Result Type
`strcpy`	Makes a copy of `source`, a string, in the character array accessed by `dest`: `strcpy(s1, "hello");`	`char *dest` `const char *source`	`char *` `h e l l o \0 ? ? ...`
`strncpy`	Makes a copy of up to `n` characters from `source` in `dest`: `strncpy(s2, "inevitable", 5)` stores the first five characters of the source in `s1` and does NOT add a null character.	`char *dest` `const char *source` `size_t`† `n`	`char *` `i n e v i ? ? ...`
`strcat`	Appends `source` to the end of `dest`: `strcat(s1, "and more");`	`char *dest` `const char *source`	`char *` `h e l l o a n d   m o r e \0`
`strncat`	Appends up to `n` characters of `source` to the end of `dest`, adding the null character if necessary: `strncat(s1, "and more", 5);`	`char *dest` `const char *source` `size_t`† `n`	`char *` `h e l l o a n d   m \0 ?`
`strcmp`	Compares `s1` and `s2` alphabetically; returns a negative value if `s1` should precede `s2`, a zero if the strings are equal, and a positive value if `s2` should precede `s1` in an alphabetized list: `if (strcmp(name1, name2) == 0)...`	`const char *s1` `const char *s2`	`int`
`strncmp`	Compares the first `n` characters of `s1` and `s2` returning positive, zero, and negative values as does `strcmp`: `if (strncmp(n1, n2, 12) == 0) ...`	`const char *s1` `const char *s2` `size_t`† `n`	`int`
`strlen`	Returns the number of characters in `s`, not counting the terminating null: `strlen("What")` returns `4`.	`const char *s`	`size_t`
`strtok`	Breaks parameter string source into tokens by finding groups of characters separated by any of the delimiter characters in `delim`. First call must provide both `source` and `delim`. Subsequent calls using NULL as the source string find additional tokens in original source. Alters source by replacing first delimiter following a token by '\0'. When no more delimiters remain, returns rest of source. For example, if `s1` is `"Jan.12,.1842"`, `strtok(s1,".,")` returns `"Jan"`, then `strtok (NULL,".,")` returns `"12"` and `strtok(NULL,".,")` returns `"1842"`. The memory in the right column shows the altered `s1` after the three calls to `strtok`. Return values are pointers to substrings of `s1` rather than copies.	`const char *source` `const char *delim`	`char *` `J a n \0 1 2 \0 1 8 4 2 \0`

† `size_t` is an unsigned integer

Functions whose names are in color change a string value, but do not take a parameter indicating the size of the destination string or the size of the string to copy, and therefore do not provide the programmer with a means of preventing buffer overflow.

CS 210 **October 25, 2016**
Strings

A palindrome is a word or a sentence which reads the same forward or backward. For example, *mom* is a palindrome as are all of the following (from Wikipedia):
redivider, noon, civic, radar, level, rotor, kayak, reviver, racecar, redder, madam, and refer.

Write a program to input a sentence on one line from the user and determine if the sentence is a palindrome. Your program should print whether or not the input sentence is a palindrome.

Turn in a hard copy of your source code.

CS 210 **October 27, 2016**
Day 16

Go over palindrome problem
A palindrome is a word or a sentence which reads the same forward or backward. For example, *mom* is a palindrome as are all of the following (from Wikipedia):
redivider, noon, civic, radar, level, rotor, kayak, reviver, racecar, redder, madam, and refer.

Write a program to input a sentence on one line from the user and determine if the sentence is a palindrome. Your program should print whether or not the input sentence is a palindrome.

Turn in a hard copy of your source code.

SOLUTION 1
```c
#include<stdio.h>
#include<string.h>
#include<ctype.h>
/* This version compares characters in a sentence,
    first with last, next with next to last, etc. to
    determine if the sentence is palindomic.
*/
int main()
{
    char line[80];
    int i, len;
    int IsPalindrome = 1;
    printf("Input a one line sentence... \n");
    gets(line);
    len = strlen(line);
    for(i=0;i<len;i++)
       line[i] = tolower(line[i]);
    for(i=0;i<len/2;i++)
    {
        if(line[i] != line[len-i-1])
           IsPalindrome = 0;
    }
    if(IsPalindrome)
        printf("%s is a palindrome.\n", line);
    else
        printf("%s is NOT a palindrome.\n", line);
}
```

SOLUTION 2

```
/* This version creates a new reversed sentence and
    compares it to the original to determine if the
    sentence is palindromic.
*/
int main()
{
    char line[80];
    char pal[80];
    int i, len;
    int IsPalindrome = 1;
    printf("Input a one line sentence... \n");
    gets(line);
    len = strlen(line);
    for(i=0;i<len;i++)
       line[i] = tolower(line[i]);
    for(i=0;i<len;i++)
    {
         pal[i] = line[len-i-1];
    }
    pal[i] = '\0';
    if(strcmp(line, pal) == 0)
        printf("%s is a palindrome.\n", line);
    else
        printf("%s is NOT a palindrome.\n", line);
}
```

STRTOK – String token
The strtok string function in string.h breaks a string into tokens where tokens are arbitrary characters defined by the user. For example, if I have a string "Mud and chocolate don't mix.", I could define a token a space " ". strtok could then be used to break this string into five words separated by spaces.

Here is an example: Track through this example with the debugger.
```
#include<stdio.h>
#include<string.h>

int main ()
{
  char str[] ="- This, a sample string.";
  char *pch;
  char *context;
  const char sTokens[] = " ,.-";
  pch = strtok_s(str, sTokens, &context);
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok_s(NULL, sTokens, &context);
  }
  return 0;
}
```

Or, if you want to use the unsafe strtok:
```
#include<stdio.h>
#include<string.h>
#pragma warning(disable:4996)

int main ()
{
  char str[] ="- This, a sample string.";
  char *pch;
  char sTokens[] = " ,.-";
  pch = strtok (str, sTokens);
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok(NULL, sTokens);
  }
  return 0;
}
```

Sample 2-D array declaration.
```
int grid[10][7];  //Declare a two-D grid
```

Sample prototype for a 2D array. Only second dimension is specified
```
void MyFunction(int grid[][10]);
```

Sample function definition for passing grid.
```
void MyFunction(int grid[][10])
{
   Grid[i][j] = value;
}
```

Sample 3-D array declaration.
```
int grid[10][7][3];  //Declare a three-D grid
```

Sample prototype for a 2D array. Only second dimension is specified
```
void MyFunction(int grid[][7][3]);
```

Sample function definition for passing grid.
```
void MyFunction(int grid[][7][3])
{
   Grid[i][j][k] = value;
}
```

Initializing a 1D array
```
int d1[] = {1, 2, 3, 4, 5};
```

Initializing a 2D array
```
int d2[][4] = {{1, 2, 3, 4, 5},
               {6, 7, 8, 9, 0},
               {1, 2, 3, 4, 5},
               {6, 7, 8, 9, 0}
              };
```

Initializing a 3D array, 2 rows, 2 columns, and 5 planes.
```
int d3[][2][5] = {{{1, 2, 3, 4, 5}, {6, 7, 8, 9, 0}},
                  {{0, 9, 8, 7, 6}, {5, 4, 3, 2, 1}}
                 };
```

Example problem – student grades

```
#include<stdio.h>
void PrintAll(int grades[][4]);
void PrintStudentAvg(int grades[][4]);
void PrintExamAvg(int grades[][4]);
int main()
{
 int grades[][4] = {{99, 90, 50, 94},
                    {88, 78, 87, 95},
                    {80, 78, 70, 80},
                    {75, 76, 77, 78},
                    {69, 79, 89, 99}
                   };
 PrintAll(grades);
 PrintStudentAvg(grades);
 PrintExamAvg(grades);
}
void PrintAll(int grades[][4])
{
   int r, c;
   for(r=0;r<5;r++)
   {
       for(c=0;c<4;c++)
          printf("%d, ", grades[r][c]);
       printf("%\n");
   }
}
void PrintStudentAvg(int grades[][4])
      {int r, c, sum;
       double avg;
       for(r=0;r<5;r++)
         {sum = 0;
          for(c=0;c<4;c++)
             sum += grades[r][c];
          avg = (double)sum/4;
          printf("Student %d %6.2f\n", r, avg);
         }
      }
void PrintExamAvg(int grades[][4])
      {int r, c, sum;
       double avg;
       for(c=0;c<4;c++)
         {sum = 0;
          for(r=0;r<5;r++)
             sum += grades[r][c];
          avg = (double)sum/5;
          printf("Test %d %6.2f\n", c, avg);
         }
      }
```

Prints the following:

```
99, 90, 50, 94,
88, 78, 87, 95,
80, 78, 70, 80,
75, 76, 77, 78,
69, 79, 89, 99,
Student 0  83.25
Student 1  87.00
Student 2  77.00
Student 3  76.50
Student 4  84.00
Test 0  82.20
Test 1  80.20
Test 2  74.60
Test 3  89.20
Press any key to continue . . .
```

CS 210 **October 25, 2016**
Multidimensional Arrays

Write a program in the c-language to define a 3D array as shown:

```
int d3[][3][5] = {{{1, -2, 3, -4, 5}, {6, -7, 8, 9, 0}},
                   {{10, -9, 8, 7, 6}, {5, 4, -3, 2, 1}},
                   {{6, -9, 3, -2, 6}, {1, 8, -6, 1, 5}}
                  };
```

Your program should go through the array and change all negative numbers to zero. Print the final array to the console.

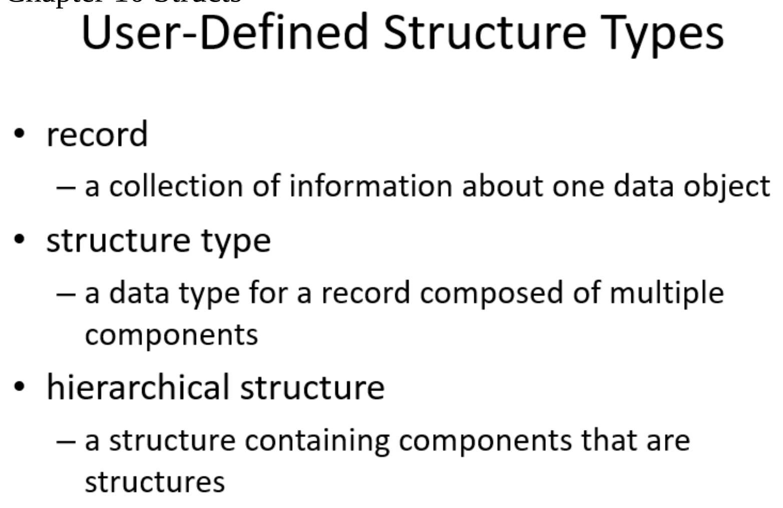
Turn in a hard copy of source code.

Sample results:

```
1, 0, 3, 0, 5,
6, 0, 8, 9, 0,
0, 0, 0, 0, 0,

10, 0, 8, 7, 6,
5, 4, 0, 2, 1,
0, 0, 0, 0, 0,

Press any key to continue . . .
```

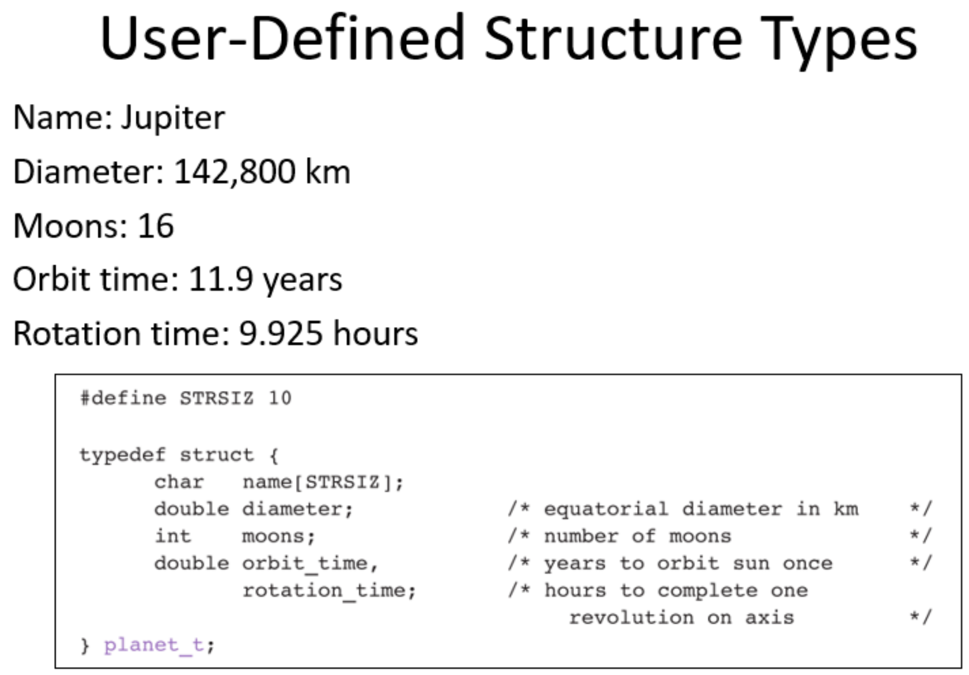
CS 210 **November 1, 2016**
Day 17

Chapter 10 Structs



User-Defined Structure Types

- record
  - a collection of information about one data object
- structure type
  - a data type for a record composed of multiple components
- hierarchical structure
  - a structure containing components that are structures



User-Defined Structure Types

Name: Jupiter

Diameter: 142,800 km

Moons: 16

Orbit time: 11.9 years

Rotation time: 9.925 hours

```
#define STRSIZ 10

typedef struct {
      char   name[STRSIZ];
      double diameter;           /* equatorial diameter in km    */
      int    moons;              /* number of moons              */
      double orbit_time,         /* years to orbit sun once      */
             rotation_time;      /* hours to complete one
                                      revolution on axis         */
} planet_t;
```

This example shows how to declare a struct and use those values in an equation.

```
#include<stdio.h>
#include<math.h>
typedef struct
{
    double x;
    double y;
}point_t;
//Note that since structs are typedefs and easily
//  confused with simple variables, they are
//  typically named with lower case letter followed
//  by _t as in point_t
int main()
{
    point_t p1, p2;
    double distance;
    p1.x = 12.5;
    p1.y = 9.2;
    p2.x = 5.3;
    p2.y = -9.1;
    distance = sqrt(pow(p1.x - p2.x, 2) + pow(p1.y-p2.y,2));
    printf("The distance from %3.2f, %3.2f to %3.2f, %3.2f is %3.2f\n",
                   p1.x, p1.y, p2.x, p2.y, distance);
    //Structs can be copied like variables
    p2 = p1;
    printf("P1 =  %3.2f, %3.2f, P2 =  %3.2f, %3.2f \n",
                 p1.x, p1.y, p2.x, p2.y);
    return 0;
}
/*Prints the following:
The distance from 12.50, 9.20 to 5.30, -9.10 is 19.67
P1 =  12.50, 9.20, P2 =  12.50, 9.20
Press any key to continue . . .
*/
```

Note that even though you can copy one struct to another as in p1 = p2 you cannot use the comparison operators on structs: **`if(p1 < p2)`** ***is illegal.***

This example show how to use a struct as a parameter passed to a function

```c
#include<stdio.h>
#include<math.h>
typedef struct
{
    double x;
    double y;
}point_t;
double FindDistance(point_t p1, point_t p2);
int main()
{
    double distance;
    point_t p1, p2;
    p1.x = 0; p1.y = 0;
    p2.x = 5; p2.y = 5;
    distance = FindDistance(p1, p2);
    printf("The distance between (%6.2f, %6.2f) and (%6.2f, %6.2f) is %6.2f\n",
                p1.x, p1.y, p2.x, p2.y, distance);
    return 0;
}
double FindDistance(point_t p1, point_t p2)
{
   double d;
   d = sqrt(pow(p1.x-p2.x, 2)+pow(p1.y-p2.y, 2));
   return d;
}
/*The distance between (  0.00,   0.00) and (  5.00,   5.00) is   7.07
Press any key to continue . . .
*/
```

Structs can also be passed by reference using * and & just like variables.

```
#define PI 3.14159265359
#include<stdio.h>
#include<math.h>
typedef struct
{
    double x;
    double y;
}point_t;
//This function returns a point p2 a distance d and
//  angle theta from p1.
void FindPoint(point_t p1, point_t *p2, double d, double theta);
int main()
{
    double d, theta;
    point_t p1, p2;
    p1.x = 0; p1.y = 0;
    d = 10; theta = 45;
    FindPoint(p1, &p2, d, theta);
    printf("The new point is at (%6.2f, %6.2f)\n",
                p2.x, p2.y);
    return 0;
}
void FindPoint(point_t p1, point_t *p2, double d, double theta)
{
    (*p2).x = d * cos(theta*PI/180) + p1.x;
    (*p2).y = d * sin(theta*PI/180) + p1.y;
}
/*Note that writing *p2.x because, according to table
10.1 p. 576 the dot operator is done before * operator.

The notation (*p2).x is awkward so there is a new symbol
that replaces it.  We can write p2 -> x instead.
*/
//void FindPoint(point_t p1, point_t *p2, double d, double theta)
//{
//    p2 -> x = d * cos(theta*PI/180) + p1.x;
//    p2 -> y = d * sin(theta*PI/180) + p1.y;
//}
```

The arrow -> operator has the awkward name of *indirect component selection operator*.

It is also possible to create an array of structs. This is the shortest distance problem from Asn 06

```c
#ifdef _MSC_VER
#define _CRT_SECURE_NO_WARNINGS
#endif
#include<stdio.h>
#include<math.h>
typedef struct
{
    double x;
    double y;
}point_t;
double FindLine(point_t p1[], int numPoints);
double DistanceBetweenPoints(point_t pt1, point_t pt2);

int main()
{
    double d;
    point_t p1[20];
    int i, totalPts, status;
    FILE *inFilep;
    inFilep = fopen("Asn06.txt", "r");
    i = 0;
    while ((status = fscanf(inFilep, " %lf,%lf", &p1[i].x, &p1[i].y)) != EOF && i < 20)
    {
        printf("%lf, %lf\n", p1[i].x, p1[i].y);
        i++;
    }
    printf("Number of items = %d\n", i);
    totalPts = i;
    fclose(inFilep);
    d = FindLine(p1, totalPts);
    printf("Shortest distance is %6.2f\n", d);
    return 0;
}
double FindLine(point_t p1[], int numPoints)
{
    int i, j;
    double shortest;
    double a;
    shortest = DistanceBetweenPoints(p1[0], p1[1]);

    for (i = 0; i<numPoints - 1; i++)
    {
        for (j = i + 1; j<numPoints; j++)
        {
            a = DistanceBetweenPoints(p1[i], p1[j]);
            if(shortest > a)
               shortest = a;
        }
    }
    return shortest;
}
double DistanceBetweenPoints(point_t pt1, point_t pt2)
{
    double d1, d2;
    d1 = pt1.x - pt2.x;
    d2 = pt1.y - pt2.y;
    return sqrt(d1*d1 + d2*d2);
}
```

CS 210 **November 1, 2016**
Structs

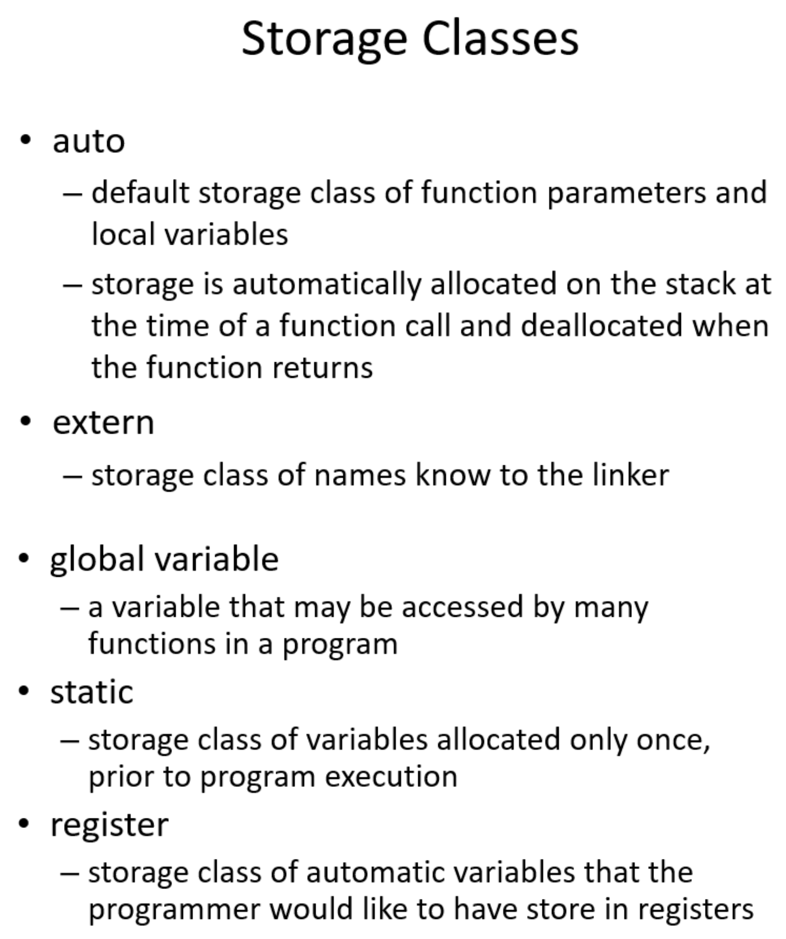
A complex number has a real part and an imaginary part. Define a struct as shown in the main program outline below. The program defines a complex number struct and three complex variable c1, c2, and c3. It calls a function called MultiplyComplex which accepts two complex arguments and returns a complex result. The function mulitplies the two complex arguments to get the result. Complex multiplication is defined as follows:
If `x = a + ib` and `y = c + id` then
`z = x * y = (a*c – b*d) + i(ad + cb)`
where `i` is $\sqrt{-1}$

```
typedef struct
{
    double real;
    double imag;
} complex_t;
//Put your function prototype here.
int main()
{
   complex_t c1, c2, c3;
   c1.real = 4; c1.imag = 3;
   c2.real = 1; c2.imag = 2;
   c3 = MultiplyComplex(c1, c2);
   printf("(%4.1f + i%4.1f) * (%4.1f + i %4.1f) = (%4.1f + i%4.1f)\n",
       c1.real, c1.imag, c2.real, c2.imag, c3.real, c3.imag);
   return 0;
}
//Put your function definition here.
```

Turn in a hard copy of your source code.

CS 210 **November 3, 2016**
Day 18

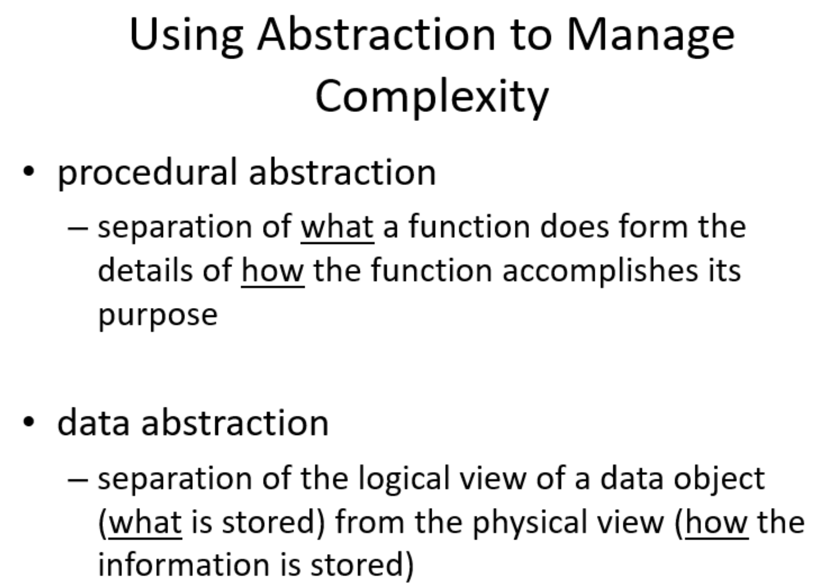
Chapter 12 Separate Compilation



Storage Classes

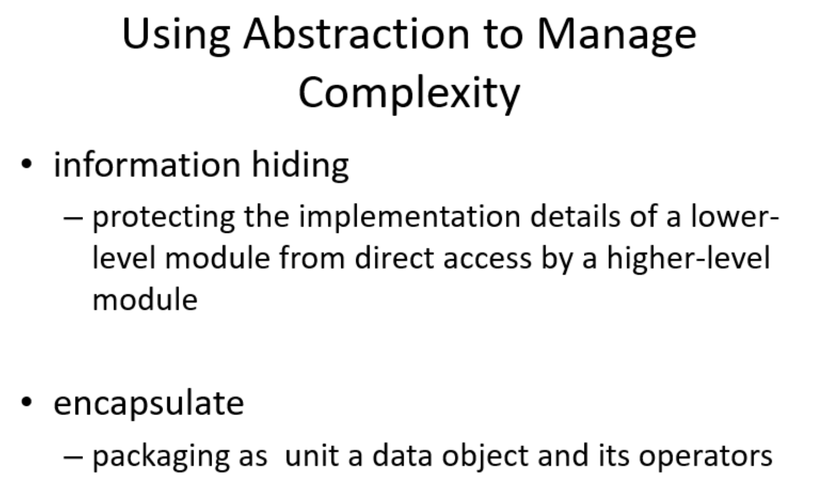
- auto
  - default storage class of function parameters and local variables
  - storage is automatically allocated on the stack at the time of a function call and deallocated when the function returns
- extern
  - storage class of names know to the linker
- global variable
  - a variable that may be accessed by many functions in a program
- static
  - storage class of variables allocated only once, prior to program execution
- register
  - storage class of automatic variables that the programmer would like to have store in registers

It is possible to write a program in multiple parts, compile these parts separately, and put them together to form a complete project. In this chapter we look at how this is done.



Using Abstraction to Manage Complexity

- procedural abstraction
  - separation of <u>what</u> a function does form the details of <u>how</u> the function accomplishes its purpose
- data abstraction
  - separation of the logical view of a data object (<u>what</u> is stored) from the physical view (<u>how</u> the information is stored)

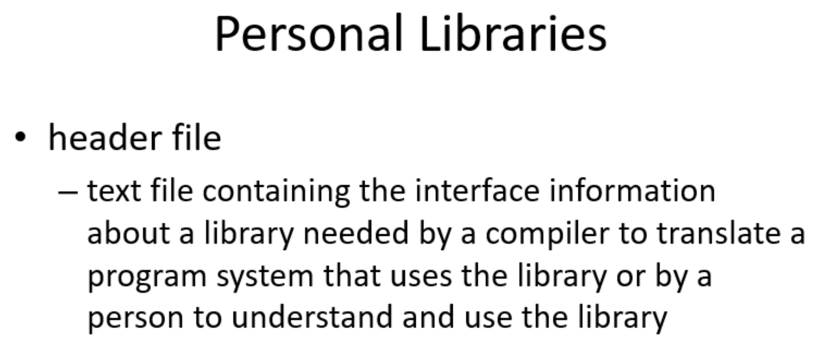


Using Abstraction to Manage Complexity

- information hiding
  - protecting the implementation details of a lower-level module from direct access by a higher-level module
- encapsulate
  - packaging as unit a data object and its operators

A good example of *abstraction* is a high level language like C. It allows you to use a computer system without knowing much at all about what is going on inside. This is not too different from learning how to drive a car without understanding much at all about the engine or transmission.

One way for you to do abstraction is to create your own library functions. For example, you might write a function which does file I/O. you could put this function in a personal library and reuse it in many programs. To anyone else looking at he program the library functions would be abstract. This is similar to what you now with stdio.h or math.h.

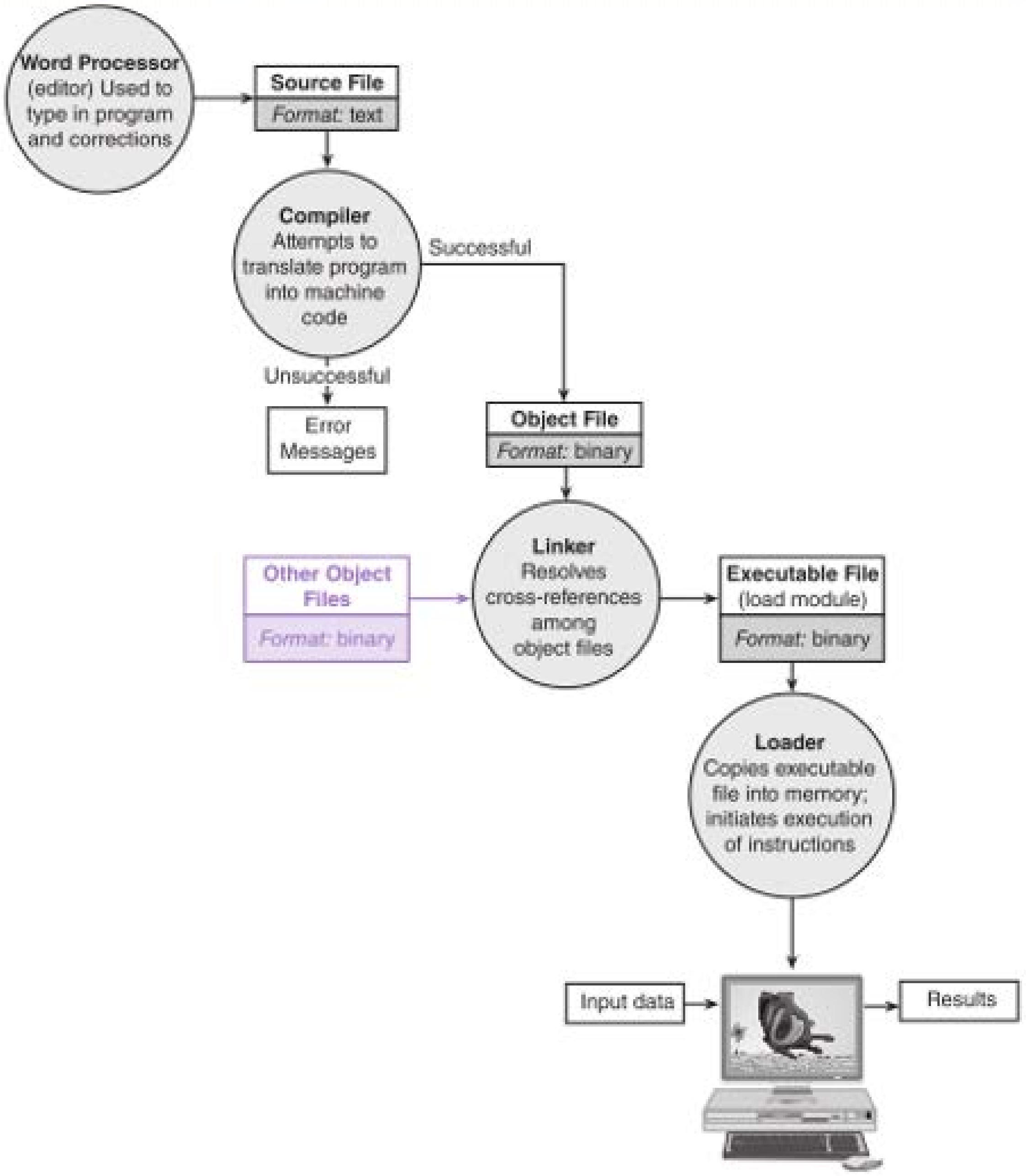


Personal Libraries

- header file
  - text file containing the interface information about a library needed by a compiler to translate a program system that uses the library or by a person to understand and use the library

The figure on the next page shows what happens when you write and execute a program.

FIGURE 12.1 Preparing a Program for Execution


Word Processor
(editor) Used to type in program and corrections
Source File
Format: text
Compiler
Attempts to translate program into machine code
Successful
Unsuccessful
Error Messages
Object File
Format: binary
Other Object Files
Format: binary
Linker
Resolves cross-references among object files
Executable File
(load module)
Format: binary
Loader
Copies executable file into memory; initiates execution of instructions
Input data
Results

There are several ways to create personal libraries. A very easy way to do this is to create a set of functions in a separate c-language file and include them in your project.

Here is a normal project (as we have been writing it). The program uses two points (x_1, y_1) and (x_2, y_2) and finds the length of a line between them as well as the slope of line from the first point to the second.

```c
#include<stdio.h>
#include<math.h>
double FindDistance(double x1, double y1, double x2, double y2);
double FindSlope(double x1, double y1, double x2, double y2);
int main()
{
    double x1, y1, x2, y2;
    double dist, slope;
    x1 = 0; y1 = 0;
    x2 = 1.0; y2 = 1.0;
    dist = FindDistance(x1, y1, x2, y2);
    slope = FindSlope(x1, y1, x2, y2);
    printf("Line from (%4.1f, %4.1f) to (%4.1f, %4.1f)\n",
                x1, y1, x2, y2);
    printf("has a length of %6.2f\n", dist);
    printf("and a slope of %6.2f\n", slope);
    return 0;
}
double FindDistance(double x1,double y1,double x2,double y2)
{return sqrt(pow(x1-x2, 2)+pow(y1-y2, 2));
}
double FindSlope(double x1,double y1,double x2,double y2)
{return (y2 - y1)/(x2 - x1);
}
```

This program can be broken into two programs – one has the main program and the second has the functions.

```c
#include<stdio.h>
#include<math.h>
extern double FindDistance(double x1,
        double y1, double x2, double y2);
extern double FindSlope(double x1,
        double y1, double x2, double y2);
int main()
{
    double x1, y1, x2, y2;
    double dist, slope;
    x1 = 0; y1 = 0;
    x2 = 1.0; y2 = 1.0;
    dist = FindDistance(x1, y1, x2, y2);
    slope = FindSlope(x1, y1, x2, y2);
    printf("Line from (%4.1f, %4.1f) to
            (%4.1f, %4.1f)\n",
               x1, y1, x2, y2);
    printf("has a length of %6.2f\n", dist);
    printf("and a slope of %6.2f\n", slope);
    return 0;
}
```

LineNormal2.c

```c
#include<math.h>

double FindDistance(double x1,
              double y1,double x2,double y2)
{return sqrt(pow(x1-x2, 2)+pow(y1-y2, 2));
}

double FindSlope(double x1,double y1,
              double x2,double y2)
{return (y2 - y1)/(x2 - x1);
}
```

LineNormalLib.c

Note that we have two source programs. The main program no longer contains the function definitions. Instead it contains only the function prototypes and these are declared **`extern`**.

The second program called LineNormalLib.c contains the function definitions.

You can also put the function prototypes in a separate header file. To do this right click on Header Files in the solution explorer and add a header file to the program.

```c
#include<stdio.h>
#include<math.h>
#include "LineNormalLib.h"
int main()
{
    double x1, y1, x2, y2;
    double dist, slope;
    x1 = 0; y1 = 0;
    x2 = 1.0; y2 = 1.0;
    dist = FindDistance(x1, y1, x2, y2);
    slope = FindSlope(x1, y1, x2, y2);
    printf("Line from (%4.1f, %4.1f) to
       (%4.1f, %4.1f)\n", x1, y1, x2, y2);
    printf("has a length of %6.2f\n",
                             dist);
    printf("and a slope of %6.2f\n",
                             slope);
    return 0;
}
```

LineNormal2.c

```c
#include<math.h>

double FindDistance(double x1,
          double y1,double x2,double y2)
{return sqrt(pow(x1-x2, 2)+pow(y1-y2, 2));
}
double FindSlope(double x1,
          double y1,double x2,double y2)
{return (y2 - y1)/(x2 - x1);
}
```

LineNormalLib.c

```c
#pragma once
extern double FindDistance(double x1,
         double y1, double x2, double y2);
extern double FindSlope(double x1,
         double y1, double x2, double y2);
```

LineNormalLib.h

In this version there are three files. LineNormal2.c is the main program. LineNormalLib.c contains our "library" functions, and LineNormalLib.h has the prototypes which are all declared **`extern`**.

The compiler will place the header file information in the main program and compile it. It will also compile, separately, the program with the function definitions. The linker will fill in the addresses so that the two programs work together.

A second way to create a personal library of functions is to write and compile a set of functions in a separate project. You can then compile this project, add it to a project that has your main code, and add in a header file.

To do this you need to add the library project to your main project, include the header file, and tell the main project where your library project is located.

This is explained in detail in SeparateCompilationNotes.pdf located on the web site.

Arguments to function main

- command-line arguments
  - options specified in the statement that activated a program

```
int
main(int  argc,     /* input - argument count (including
                       program name)                          */
     char *argv[]) /* input - argument vector                 */
```

Run the .exe file for this program

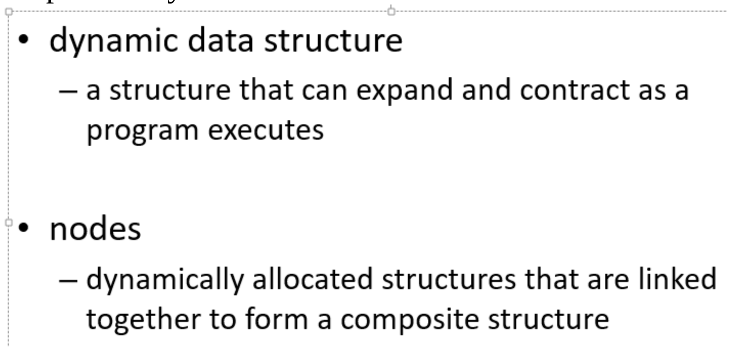
```
#include<stdio.h>
int main(int argc, char *argv[])
{
    char s2[80];
    printf("Number of arguments %d\n", argc);
    printf("Argument 0 is %s\n", argv[0]);
    printf("Argument 1 is %s\n", argv[1]);
    scanf_s("%s", s2, sizeof(s2)); //Requires buffer size
}
/*
c:\>ArgsToMain abc
Number of arguments 2
Argument 0 is ArgsToMain
Argument 1 is abc
*/
```

CS 210 **November 3, 2016**
Separately Compiled Programs

Complete the example in SeparateCompilationNotes.pdf on the web site.

CS 210 **November 8, 2016**
Day 19

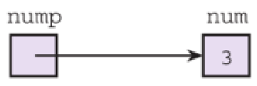
Chapter 13 Dynamic Allocation

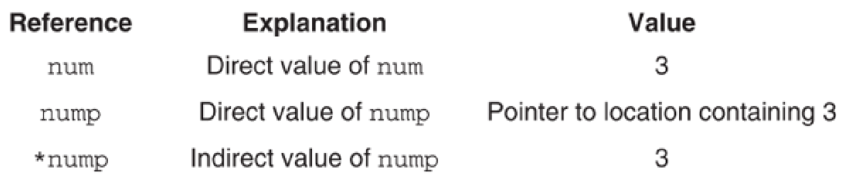


- dynamic data structure
  - a structure that can expand and contract as a program executes
- nodes
  - dynamically allocated structures that are linked together to form a composite structure

Review of pointers.

Pointers to variables





Reference	Explanation	Value
num	Direct value of num	3
nump	Direct value of nump	Pointer to location containing 3
*nump	Indirect value of nump	3

```
int num;              //num is an integer
int *nump;            //nump is a pointer to an integer
nump = &num;          //Set nump to the address of num
*nump = 3;            // *nump is used to indirectly access num
```

Pointers to Functions
We can also have pointers to functions. We can pass a function as an argument to another function. In this case the function name is represented as a pointer to a function.

The program below passes two different functions to another function for evaluation. The functions Square and Cube are passed as functions to a third function called Evaluate. In this case F is just a pointer to the function being passed.

The C-compiler will also accept
```
void Evaluate(double (*F)(double d),double d1,double d2)
```

if you want to make it more explicit that you are passing a pointer to a function.

```c
#include<stdio.h>
/* This program passes a function as a parameter
     to another function for evaluation.
*/
double Square(double d);
double Cube(double d);
void Evaluate(double F(double d), double d1, double d2);
int main()
{
   double d1 = 9;
   double d2 = 12;
   Evaluate(Square, d1, d2);
   Evaluate(Cube, d1, d2);
}
void Evaluate(double F(double d),double d1,double d2)
{
    printf("F(%5.1f) = %5.1f\n", d1, F(d1));
    printf("F(%5.1f) = %5.1f\n", d2, F(d2));
}

double Square(double d)
{
    return d*d;
}
double Cube(double d)
{
    return d*d*d;
}
```

Pointers to Arrays and Strings
When we declare an array:
`double num[12];`

The C-compilers interprets the array name as the address of the first element in the array. In this case `num` is a pointer to the array. By default arrays are passed by reference.

Both of these are syntactically correct and produce the same result

```c
#include<stdio.h>
void MyFunction(int a[], char c[]);
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    char c[] = "Hello Mom!";
    MyFunction(a, c);
}

void MyFunction(int a[],char c[])
{
    int i;
    for(i=0;i<sizeof(a);i++)
       printf("%d, ", a[i]);
    printf("\n");
    for(i=0;c[i] != 0;i++)
       printf("%c", c[i]);
    printf("\n");
}
```

```c
#include<stdio.h>
void MyFunction(int *a, char *c);
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    char c[] = "Hello Mom!";
    MyFunction(&a[0], &c[0]);
}
void MyFunction(int *a,char *c)
{
    int i;
    for(i=0;i<sizeof(a);i++)
       printf("%d, ", a[i]);
    printf("\n");
    for(i=0;c[i] != 0;i++)
       printf("%c", c[i]);
    printf("\n");
}
```

Dynamic Memory Allocation

When you compile a program the compiler calculates the amount of memory space your program will need – including the stack space and the space for all of the variables. When your program is loaded the needed memory is allocated by the operating system. This is a static allocation for the duration of your program.

If your program tries to redimension an array or it goes outside of an array boundary, chances are good that it will get outside of its allocated memory. This will trigger an operating system fault and your program will be halted.

To allow a user to redimension an array, for example we need a way to make a call to the operating system as the program is running and request additional memory. This is called a dynamic allocation since it occurs after the program has started running.

In C there is a special function called *malloc* which does dynamic memory allocation. The syntax is:

```c
malloc(sizeof(int));
```

This function returns a pointer to the memory block allocated – in this case just enough for one integer. In practice we would write:

```c
int *nump;
nump = malloc(sizeof(int));
```

The memory allocated is located in the systems read/write memory in a location referred to as the *heap*.

malloc can be used on any built-in or user defined type (i.e. structures)

```
#include<stdio.h>
#include<stdlib.h>   //needed for malloc
int main()
{
    int i = 5;
    int *iPtr;
    iPtr = (int *)malloc(sizeof(int));
    *iPtr = i + 9;
    printf("i = %d, *iPtr = %d\n ", i, *iPtr);
}
Prints:
i = 5, *iPtr = 14
Press any key to continue . . .
```

Or we can use malloc with a typedef

```
typedef struct
{
   double x;
   double y;
}point_t;
#include<stdio.h>
#include<stdlib.h>   //needed for malloc
int main()
{
    int i = 5;
    int *iPtr;
    iPtr = (int *)malloc(sizeof(int));
    *iPtr = i + 9;
    printf("i = %d, *iPtr = %d\n ", i, *iPtr);
    point_t *p1;
    p1 = (point_t *)malloc(sizeof(point_t));
    (*p1).x = 5.4;
    (*p1).y = 9.1;
    //Alternatively we can write
    p1 -> x = 5.4;
    p1 -> y = 0.1;
}
```

Note that `(*p1).x` is the same notation as `p1 -> x`

Dynamic Array Allocation
It is a little unusual to dynamically allocate a single variable or even a single typedef. The common occurrence is the dynamic allocation of an array.

C uses the function `calloc` (continuous memory allocation) to allocate arrays.
The syntax for creating an array of 20 ints is:
```
int *aPtr;
aPtr = (int *)calloc(20, sizeof(int));
```

Here is an example which asks for an array size, creates an array of that size using calloc, and prints the array forward and backward.
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i, aSize;
    int *a;
    printf("Enter the array size... ");
    scanf_s("%d", &aSize);
    a = (int *)calloc(aSize, sizeof(int));
    for(i=0;i<aSize;i++)
       a[i] = i;
    for(i=0;i<aSize;i++)
       printf("%d, %d\n", a[i], a[aSize-i-1]);
}
/*Prints the following
Enter the array size... 10
0, 9
1, 8
2, 7
3, 6
4, 5
5, 4
6, 3
7, 2
8, 1
9, 0
Press any key to continue . . .
```

Returning memory to the heap
A function called free will release allocated memory back to the heap so it can be used elsewhere.
```
free(a);  //deallocates the array in the program example above
```

CS 210 **November 8, 2016**
Dynamic Memory Allocation

The following program reads a txt file named MyData.txt and prints it to the console.
Part 1:
Copy the file "MyData.txt" from the web site and place it in the same directory as your c-code source file. Run the program below and verify that it can read and print the data in the file.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i, status, dataIn, err;
    FILE *inp;  //Declare pointers to in file
    err = fopen_s(&inp, "MyData.txt", "r");
    status = fscanf_s(inp, "%d", &dataIn);
       //read the file and print to console
    while(status == 1)
       {
              printf("%d\n", dataIn);
        status = fscanf_s(inp, "%d", &dataIn);
       }
    fclose(inp);
}
```

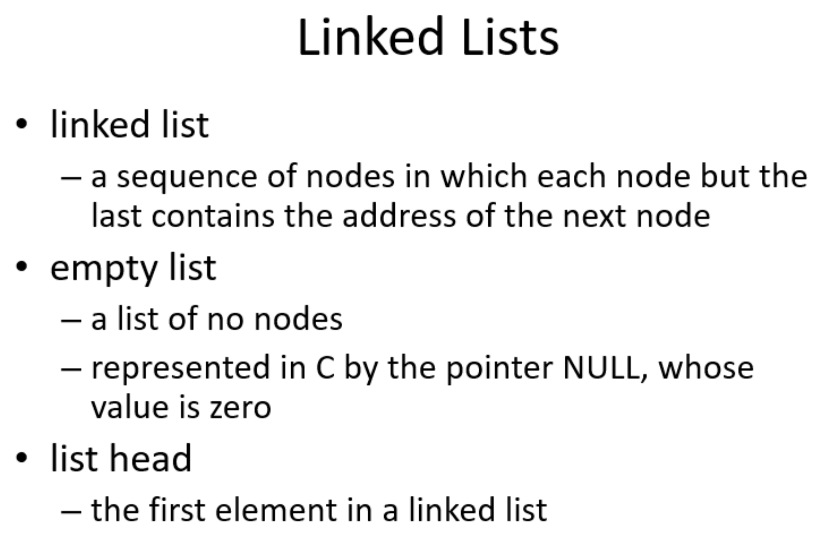
Part 2:
Modify the program above to read the first line of the file which contains the number of entries that follow. Use `calloc` to allocate an integer array to hold the data (not including the first line). Print the data from the array in reverse order on a single line with each data item separated by a comma.

Allow your instructor to verify that your program works correctly and turn in a hard copy of your source file.

CS 210 **November 10, 2016**
Day 20

Chapter 13 Stacks, Queues, and Linked Lists

Linked Lists
A linked list is a data structure made up of nodes in which each node holds some data along with a pointer to the next node in the list.

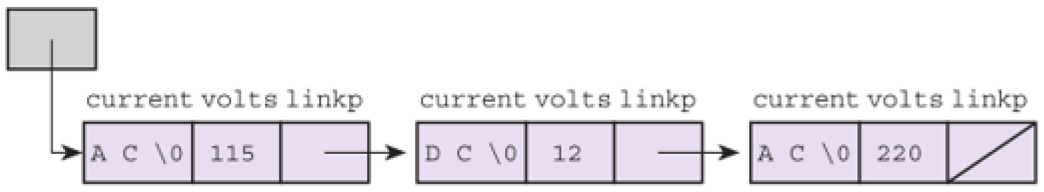


Linked Lists

- linked list
  - a sequence of nodes in which each node but the last contains the address of the next node
- empty list
  - a list of no nodes
  - represented in C by the pointer NULL, whose value is zero
- list head
  - the first element in a linked list

In C we create a linked list using structs. For example

```
typedef struct node_s
   {char current[3];
    int volts;
    struct node_s *linkp;
   }node_t;
```



```c
/*We would like to write this typedef as:
typedef struct
   {char current[3];
    int volts;
    struct node_t *linkp;
   }node_t;
but when the compiler puts this together using
struct node_t *linkp the node_t has not yet been
defined.  We use node_s as a label as shown below to
get around this.
*/
typedef struct node_s
   {char current[3];
    int volts;
    struct node_s *linkp;
   }node_t;
#include<stdio.h>
#include<string.h>
int main()
{
   node_t n1, n2, n3;        //create 3 nodes
   node_t *n1p, *n2p, *n3p; //and 3 node pointers
   n1p = &n1;                //set up the pointers
   n2p = &n2;                // to point to the nodes
   n3p = &n3;
   n1p -> linkp = n2p;   //node 1 points to node 2
   n2p -> linkp = n3p;   //node 2 points to node 3
   n3p -> linkp = NULL;  //node 3 is last
   strcpy(n1p->current, "AC"); //put values in nodes
   n1p->volts = 120;
   strcpy(n2p->current, "DC");
   n2p->volts = 240;
   strcpy(n3p->current, "AC");
   n3p->volts = 440;
   //both of these access node 2 volts
   printf("%d\n", n2p->volts);
   printf("%d\n", n1p->linkp->volts);

   return 0;
}
```

Advantages of Linked Lists

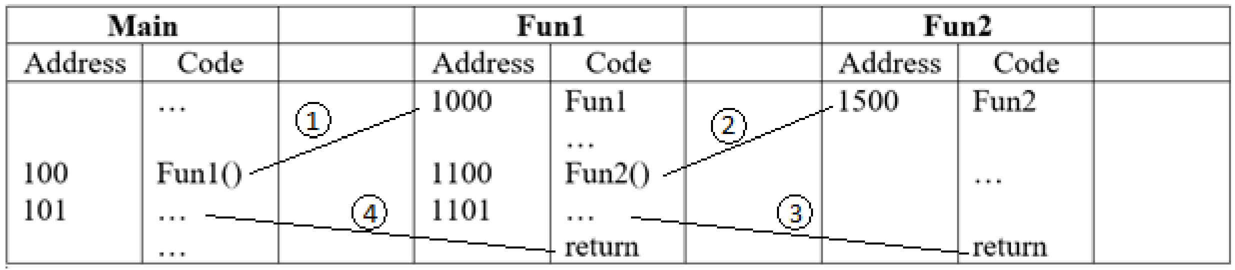
- It can be modified easily.
- The means of modifying a linked list works regardless of how many elements are in the list.
- It is easy to add or delete an element.

Stack
A stack is a first in last out buffer (or a last in first out buffer).

Nearly every computer CPU implements a stack structure in hardware with a stack pointer register on the CPU. The stack is used to save return addresses for function calls as well as pass parameters.

There are two main operators for a stack:
push – stores data on the stack and increases the stack pointer by 1
pop – decreases the stack pointer by 1 and returns the value on top of the stack.

A stack can be implements using an array (see pp. 400-401) or it can be done using a linked list (see pp. 727-731)



4→ 0 →	NULL	Top of stack
3→ 1→	101	
2→	1101	

The function calls at 1 and 2 push a return address 101 and 1101 on the stack. The return at 3 returns to 1101 since it is at the top of the stack. The stack pointer is decreased by 1 and the second return goes back to 101 which is at the top of the stack.

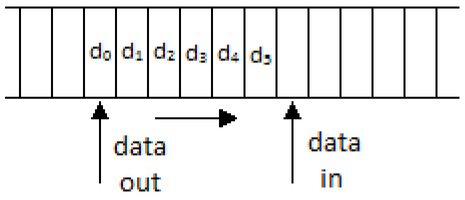
Queue
A queue is a first in first out buffer (or a last in last out buffer).

Queues are not as common or as useful as stacks but they can be used for modelling simulations or in serial data buffers.

A queue can be implemented using an array or it can be done with linked lists (see pp. 731-737).
Like stacks, queues have two main operators.
Add – places a new item in the queue at the next free location and increases the input pointer.
Remove – takes an item out of the queue and increases the output pointer.

If the input and output pointers are equal the queue is empty.

A queue can be circular. This works as long as the total buffer size is larger than the amount of data to be stored at any one time.



The data item d_0 went into the queue first and d_5 went into the queue last.