BLG 102E Introduction to Scientific Computing and Engineering

SPRING 2025

WEEK 10

ISTANBUL TECHNICAL UNIVERSITY

Dynamic Memory Allocation

Dynamic Memory Allocation

In many programming situations, you know you will be working with several values.

You would normally use an array for this situation.

Dynamic Memory Allocation

But suppose you do not know
beforehand
how many values you need.

So now can you use an array?

Dynamic Memory Allocation

The size of a *static* array must be known when you define it.

Dynamic Memory Allocation

To solve this problem, you can use

dynamic allocation.

Dynamic Memory Allocation

To use dynamic arrays, you ask the C run-time system to create new space for an array whenever you need it.

This is at RUN-TIME?

On the fly?

Arrays on demand!

Dynamic Memory Allocation

Where does this memory for my
on-demand arrays come from?

The OS *keeps*
a *heap*

Dynamic Memory Allocation

To ask for more memory,
say a **`double`**, you use the **`malloc()`** function
along with **`sizeof`** operator to get the memory
size in bytes to keep the given type.

```
(double*) malloc(sizeof(double))
```

the runtime system seeks out room for
a **`double`** on the heap, reserves it for your use
and returns a pointer to it.

This **`double`** location does not have a name.

(this is run-time)

Dynamic Memory Allocation

To request a dynamic array you use the same **`malloc`** function with some looks-like-an-array things added:

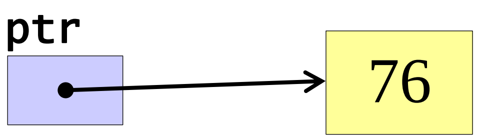
```
(double*) malloc(n * sizeof(double))
```

where **`n`** is the number of **`double`**s you want and, again, you get a pointer to the array.

Dynamic Memory Allocation

- To dynamically allocate memory **at run-time**, the *malloc()* and *sizeof()* built-in functions are used which are defined in <stdlib.h>

```
int  *ptr;                               // Pointer declaration
ptr = malloc ( sizeof (int) );           // Allocate 4 bytes and get the address
//  ptr = malloc ( 4 );                  // Same as above
*ptr = 76;                               // Changes data content to 76
```



- The allocated memory location is unnamed (generic), which can be accessible thru the ptr only.

- In order to delete a dynamically allocated memory, the ***free()*** built-in function can be used. (De-allocation)
  Example: **free(ptr);**

Dynamic Memory Allocation

You need a pointer variable to hold the pointer you get:

```
double* account_pointer =
     (double*) malloc(sizeof(double));
double* account_array =
     (double*) malloc(n * sizeof(double));
```

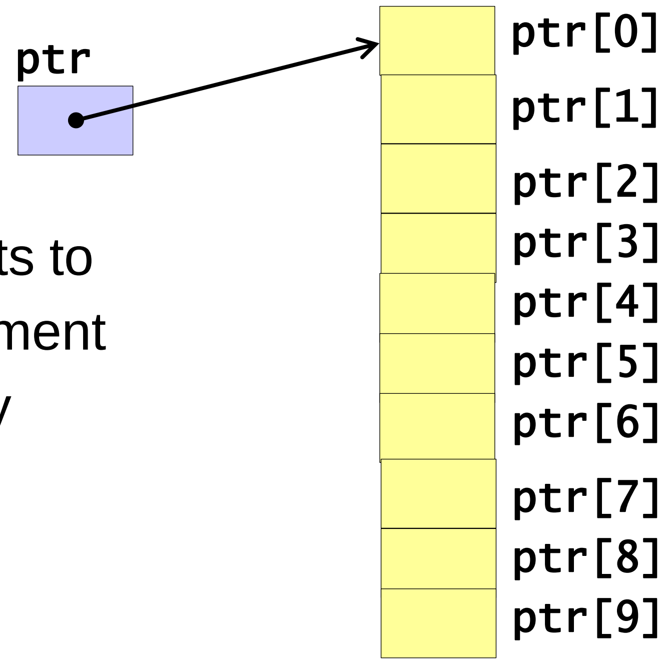
Now you can use **`account_array`** as an array.

Array/pointer duality
lets you use the array notation
`account_array[i]` to access the **`i`**th element.

Example: Allocating an Array Dynamically

- To allocate an array dynamically, multiply the sizeof with the number of elements.

```
int  *ptr;
ptr = malloc ( 10 * sizeof (int) ); // Allocate 40 bytes and get the address
//  ptr = malloc ( 40 );             // Same as above
```



ptr points to first element of array

sizeof() Function

- `sizeof`
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if `sizeof( int )` equals `4` bytes, then

```
int  Array[10];
printf( "%d", sizeof(Array) );
```

    - will print `40` (10x4 = 40)

- `sizeof` can be used with
  - Variable names
  - Type name
  - Constant values

Sizeof Values for Data Types

```
sizeof (char)           = 1
sizeof (int)            = 4
sizeof (float)          = 4
sizeof (double)         = 8
```

```
sizeof (char *)         = 4
sizeof (int *)          = 4
sizeof (float *)        = 4
sizeof (double *)       = 4
```

All kind of pointers are 4 bytes

Example: Sizeof for an array

- Program uses the sizeof() function for an array.

```
/* Sizeof operator when used on an array name
   returns the number of bytes in the array. */
#include <stdio.h>

int main()
{
   float array[ 20 ]; // create array

   printf( "The number of bytes in the array is %d \n", sizeof(array) );
   printf( "The number of elements is %d \n", sizeof(array) / sizeof(int) );

} // end main
```

Program Output

```
The number of bytes in the array is 80
The number of elements is 20
```

Dynamic Memory Allocation

When your program no longer needs the memory that you asked for with the **`malloc`** function, you must return it to the heap using the **`free`** function.

```
free(account_pointer);
free(account_array);
```

Dynamic Memory Allocation

After you delete a memory block,
you can no longer use it.
The OS is very efficient – and quick– “your” storage space may already be used elsewhere.

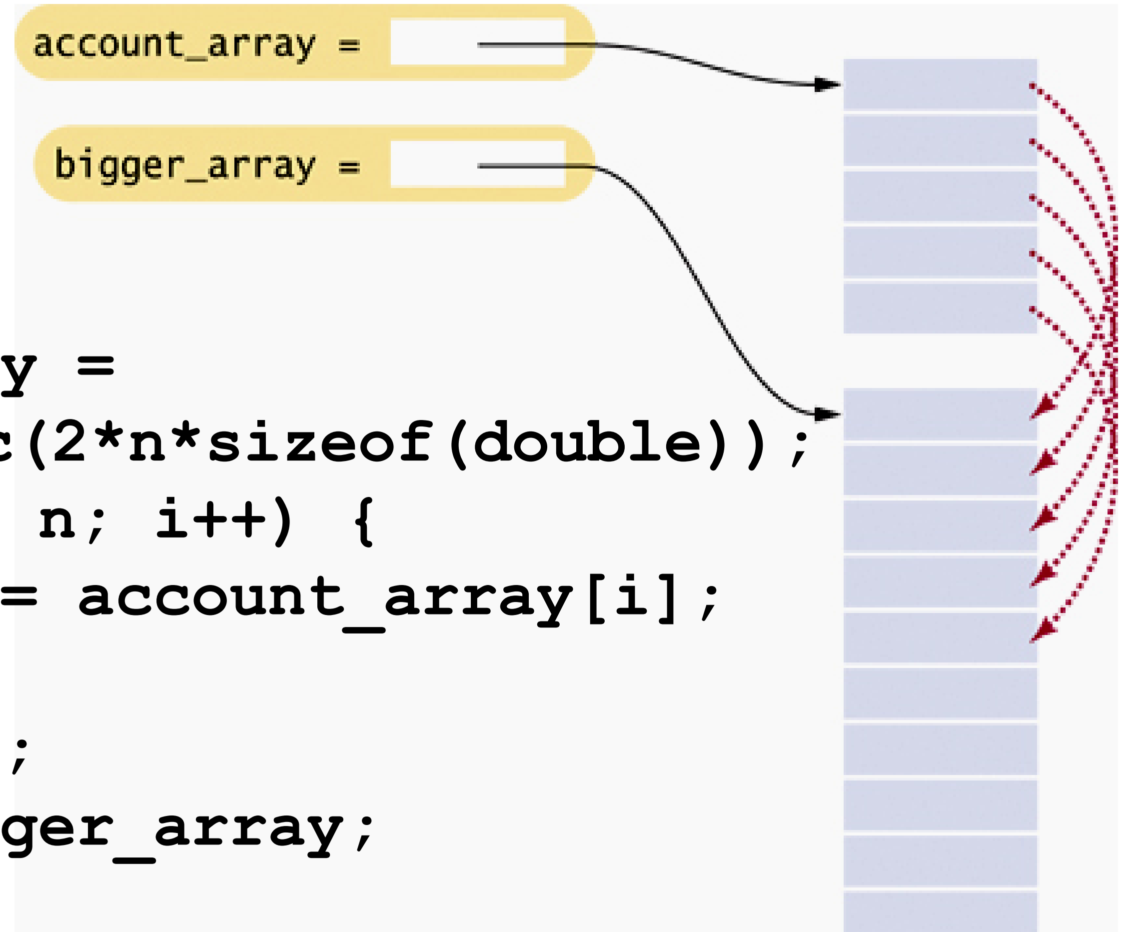
```
free(account_array);
account_array[0] = 1000;
        // NO! You no longer own the
        // memory of account_array
```

Dynamic Memory Allocation

Unlike static arrays,
which you are stuck with after you create them,
you can change the size of a dynamic array.

Make a new, improved, bigger array
and copy over the old data – but remember
to delete what you no longer need.

Dynamic Memory Allocation – Resizing an Array




```
double* bigger_array =
     (double*)malloc(2*n*sizeof(double));
for (int i = 0; i < n; i++) {
   bigger_array[i] = account_array[i];
}
free(account_array);
account_array = bigger_array;
n = 2 * n;
```

(**`n`** is the variable used with the array)

Dynamic Memory Allocation – THE RULES

1. **Every call to `malloc` *must* be matched by exactly one call to `free`.**

2. **Use `free` to delete arrays.
   And always assign `NULL` to the pointer after that.**

3. **Don’t access a memory block after it has been deleted.**

If you don’t follow these rules, your program can
***crash* or *run unpredictably*.**

Dynamic Memory Allocation

Syntax 7.2 Dynamic Memory Allocation

Capture the pointer in a variable.

The new operator yields a pointer to a memory block of the given type.

```
int* var_ptr = (int*) malloc(sizeof(int));
...
*var_ptr = 1000;
...
free (var_ptr);
```

Use the memory.

Delete the memory when you are done.

Use this form to allocate an array of the given size (size need not be a constant).

```
int* array_ptr = (int*) malloc(size*sizeof(int));
...
array_ptr[i] = 1000;
...
free (array_ptr);
```

Use the pointer as if it were an array.

Remember to use `free` when deallocating the array.

Common Errors Dangling Pointers

DANGLING

Dangling pointers **are when you use a pointer that has already been deleted or was never initialized.**

Common Errors Dangling Pointers

```
int* values = (int*)malloc(n*sizeof(int));

// Process values

free(values);

// Some other work
values[0] = 42;
```

Common Errors Dangling Pointers

The value in an uninitialized or deleted pointer might point somewhere in the program you have no right to be accessing.

You can create real damage by writing to the location to which it points.

Even just *reading* from that location can crash your program.

Common Errors Dangling Pointers

- **Always initialize pointer variables.**
- **If you can’t initialize them, then set them to `NULL`.**
- **Never use a pointer that has been deleted.**

Common Errors Memory Leaks

LEAKS

A *memory leak* is when use alolocate dynamic memory but you fail to free it when you are done.

Common Errors Memory Leaks

Remember Rule #1.

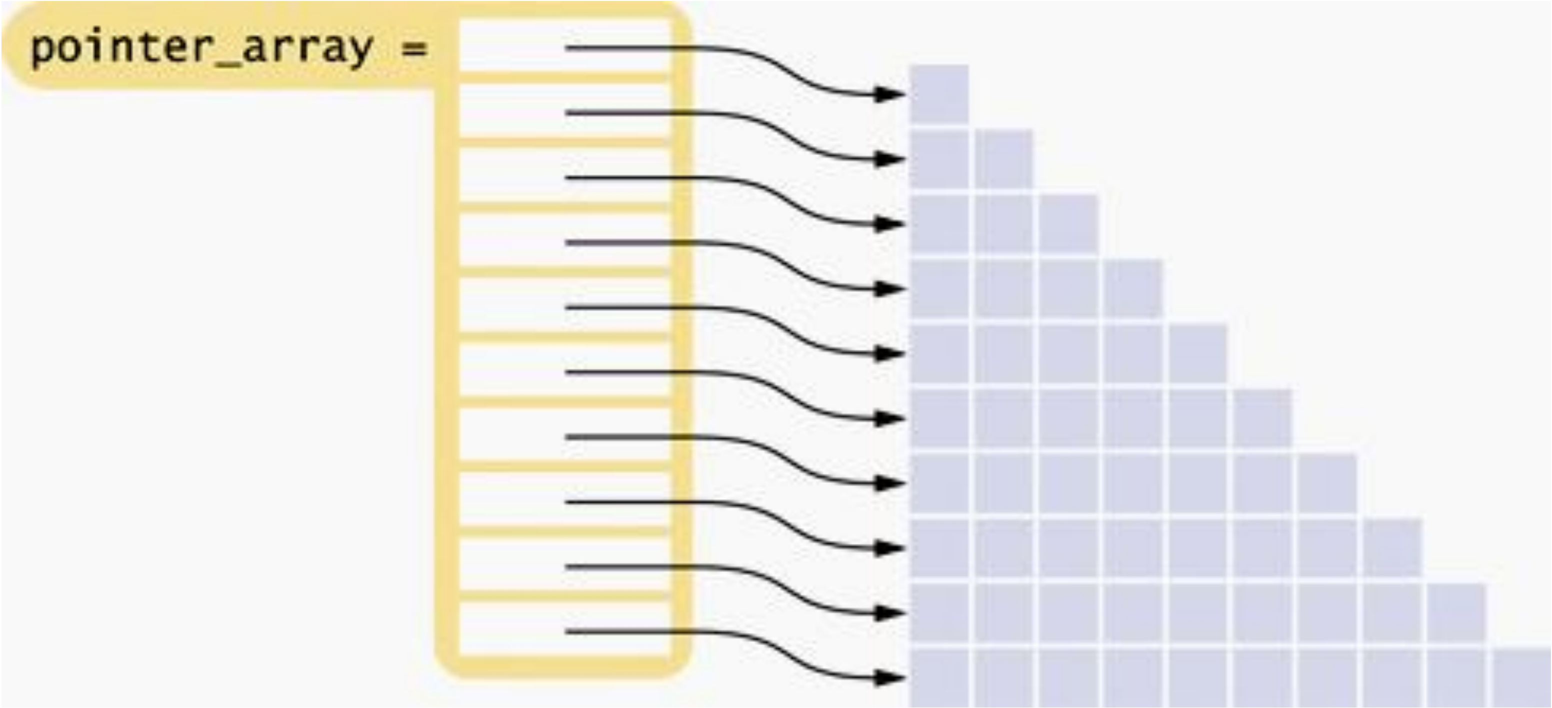
1. **Every call to `malloc` *must* be matched by exactly one call to `free`.**

And after freeing, set it to NULL so that it can be tested for danger later.

Common Errors Dangling Pointers – Serious Business

```
int* values = malloc(n * sizeof(int));

// Process values

free(values);
values = NULL;

...

if (values == NULL) ...
```

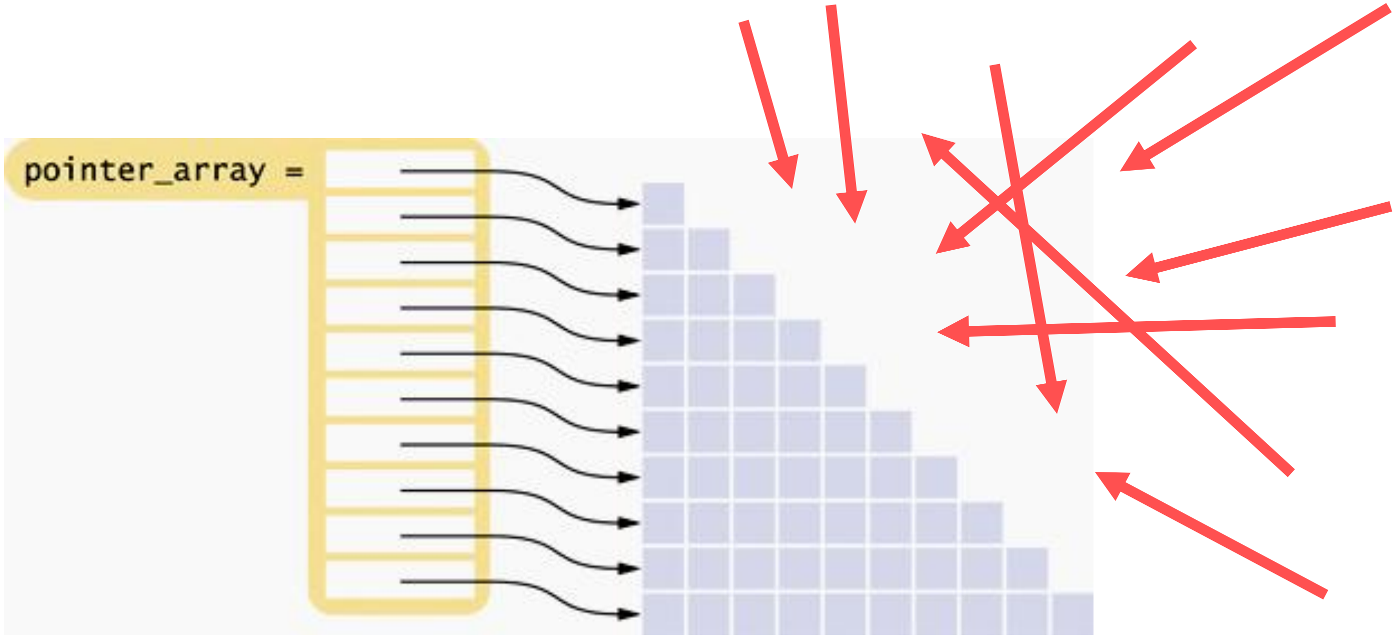
Arrays of Pointers – A Triangular Array



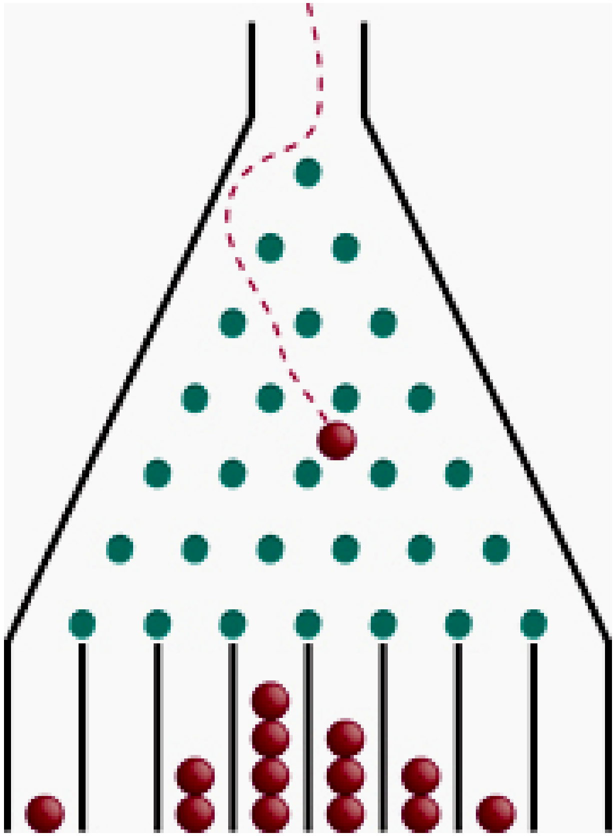
In this array, each row is a different length.

Arrays of Pointers – A Triangular Array

In this situation, it would not be very efficient
to use a two-dimensional array,
because almost half of the elements would be wasted.

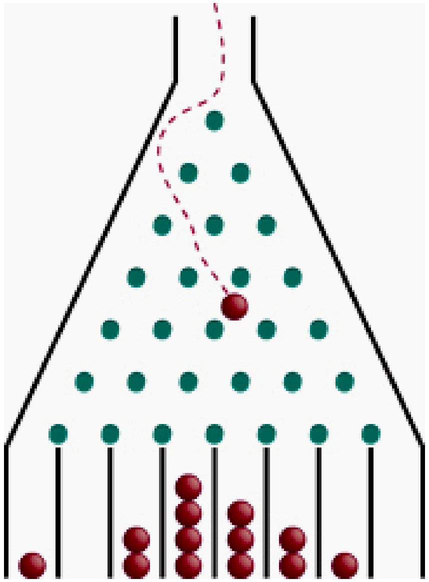


A Galton Board



A Galton Board Simulation

We will develop a program that uses a triangular array to simulate a Galton board.

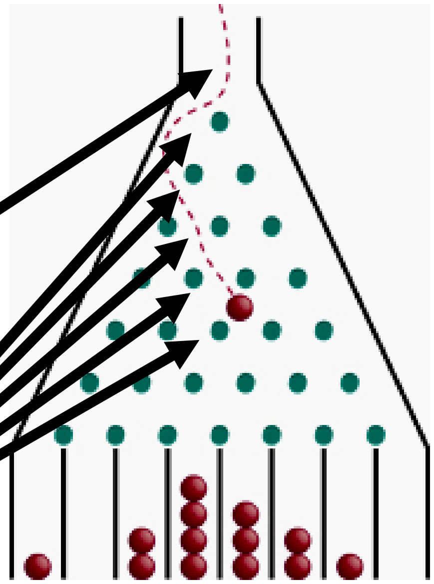


A Galton Board Simulation

A Galton board consists of a pyramidal arrangement of pegs and a row of bins at the bottom.

Balls are dropped onto the top peg and travel toward the bins.

At each peg, there is a 50 percent chance of moving left or right.



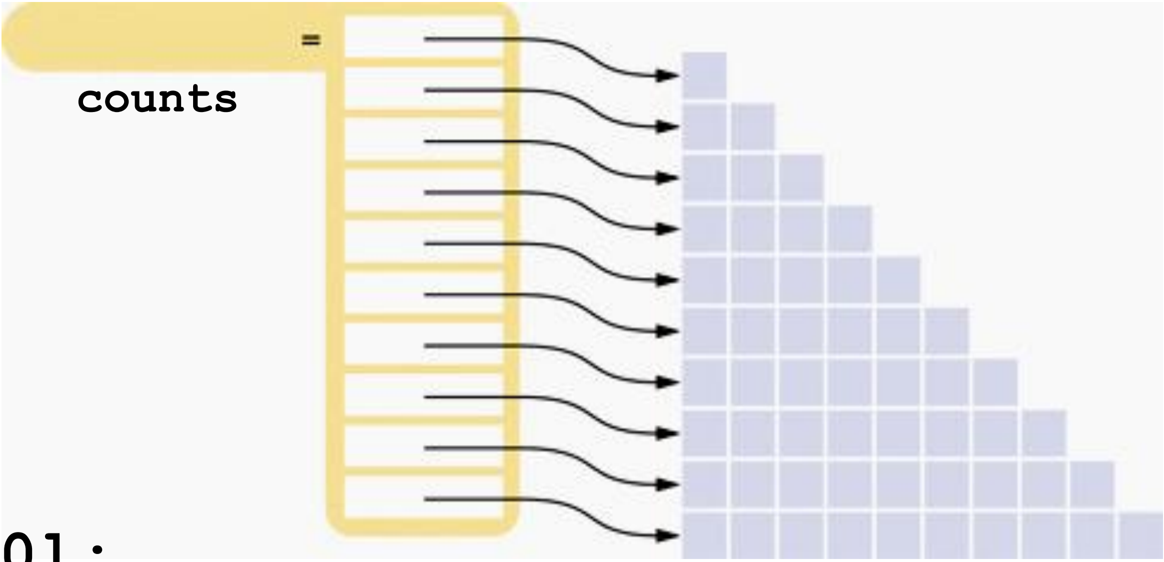
The balls in the bins approximate a bell-curve distribution.

A Galton Board Simulation

The Galton board can only show the balls in the bins, but we can do better by keeping a counter for *each* peg, incrementing it as a ball travels past it.

A Galton Board Simulation

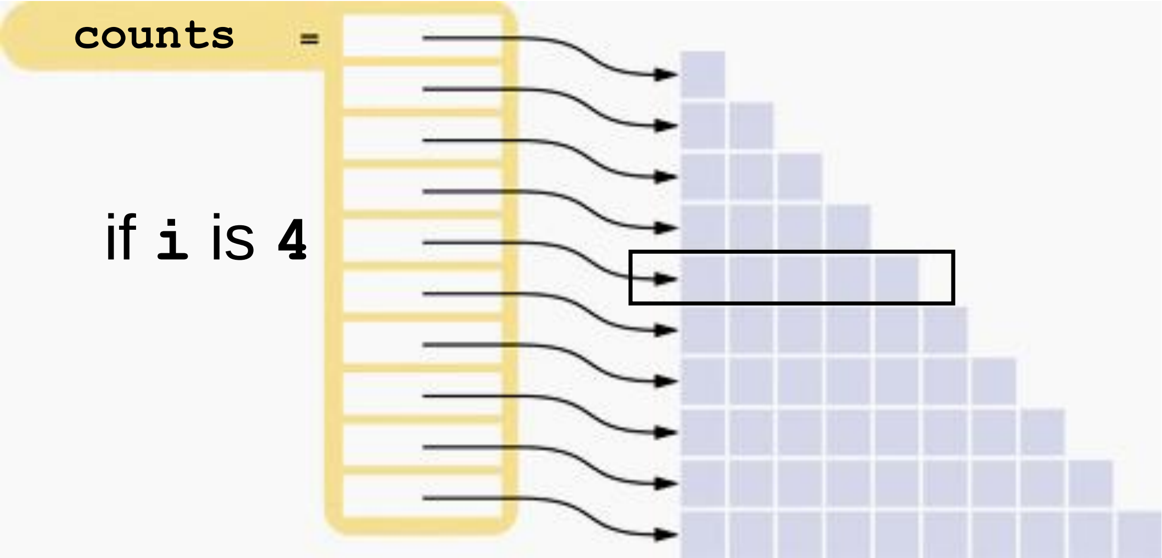
We will simulate a board with ten rows of pegs.
Each row requires an array of counters.
The following statements initialize the triangular array:




```
int* counts[10];
for (int i = 0; i < 10; i++) {
    counts[i] =
        (int*) malloc((i+1) * sizeof(int));
}
```

A Galton Board Simulation

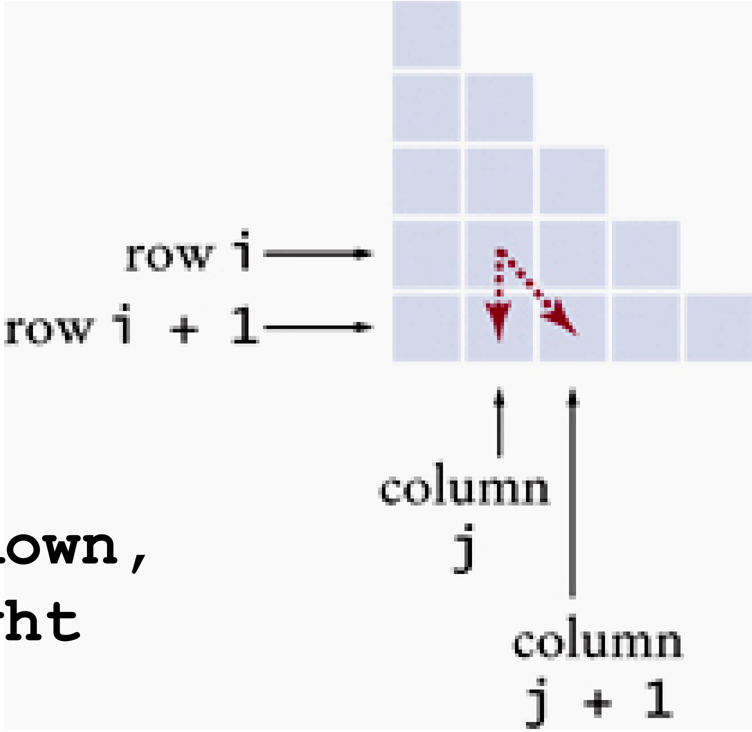
We will need to print each row:




```
// print all elements in the ith row
for (int j = 0; j <= i; j++) {
   printf("%5d", counts[i][j]);
}
printf("\n");
```

A Galton Board Simulation

We will simulate a ball bouncing through the pegs:




```
int r = rand() % 2;
// if r is even, move down,
// otherwise to the right
if (r == 1) {
    j++;
}
counts[i][j]++;
```

A Galton Board Simulation

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

int main()
{
   const int RUNS = 1000;   // Simulate 1,000 ball
   int* counts[10];
   srand(time(0));
   // allocate rows and init first two with zero
   for (int i = 0; i < 10; i++) {
      counts[i] = (int*) malloc(sizeof(int)*(i + 1));
      for (int j = 0; j <= i; j++) {
         counts[i][j] = 0;
      }
   }
```

A Galton Board Simulation

```
for (int run = 0; run < RUNS; run++) {
   // Add a ball to the top
   counts[0][0]++;
   // Have the ball run to the bottom
   int j = 0;
   for (int i = 1; i < 10; i++) {
      int r = rand() % 2;
      // If r is even, move down,
      // otherwise to the right
      if (r == 1) {
         j++;
      }
      counts[i][j]++;
   }
}
```

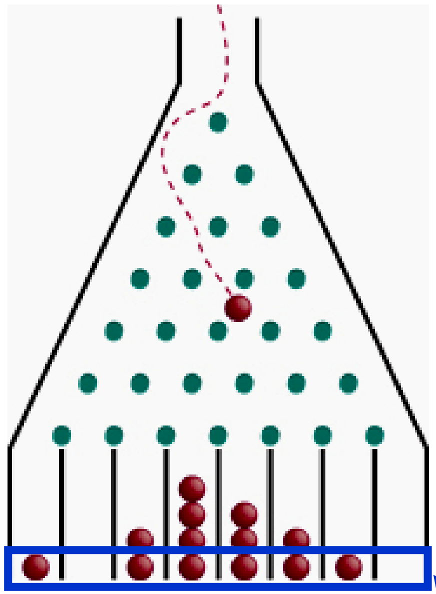
A Galton Board Simulation

```
    // Print all counts
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j <= i; j++) {
            printf("%5d",counts[i][j]);
        }
        printf("\n");
    }

    // Deallocate the rows
    for (int i = 0; i < 10; i++) {
        free(counts[i]);
    }

    return 0;
}
```

A Galton Board Simulation

This is the output
from a run of the program:

```
1000
 480 520
 241 500 259
 124 345 411 120
  68 232 365 271  64
  32 164 283 329 161  31
  16  88 229 303 254  88  22
   9  47 147 277 273 190  44  13
   5  24 103 203 288 228 113  33   3
   1  18  64 149 239 265 186  61  15   2
```



Memory Reallocation

To change/extend the size of the memory previously allocated

```
int size = 10 * sizeof(int);
int* ptr = (int*) malloc(size);
size = size * 2;
int* ptr_new = (int*) realloc(ptr, size);
```

- `realloc` changes the size of the object pointed to by `ptr` to `size`.
- The contents will be unchanged up to the minimum of the old and new sizes.
- If the new size is larger, the new space is uninitialized.
- `realloc` returns a pointer to the new space, or `NULL` if the request cannot be satisfied, in which case `ptr` is unchanged.

Memory Reallocation

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

int main()
{
   int* ptr = (int*) malloc(sizeof(int) * 2);
   int* ptr_new;

   *ptr = 10;
   *(ptr + 1) = 20;

   ptr_new = (int*) realloc(ptr, sizeof(int) * 3);
   *(ptr_new + 2) = 30;
   for(int i = 0; i < 3; i++) {
       printf("%d ", *(ptr_new + i));
   }
   free(ptr_new);
   return 0;
}
```

Strings

Strings

- Represented as arrays of `char` values.

`char` Type and Some Famous Characters

The type **`char`** is used to store an individual character.

`char` Type and Some Famous Characters

Some of these characters are plain old letters and such:

```
char yes = 'y';
char no = 'n';
char maybe = '?';
```

`char` Type and Some Famous Characters

Some are numbers masquerading as digits:

```
char theThreeChar = '3';
```

That is not the number three – it's the *character* 3.

`'3'` is what is actually stored in a disk file when you write the **`int`** 3.

Writing the variable **`theThreeChar`** to a file would put the same **`'3'`** in a file.

`char` Type and Some Famous Characters

So some characters are literally what they are:

`'A'`

Some represent digits:

`'3'`

Some are other things that can be typed:

`'C'`

`'+'`

`'+'`

but…

Some Famous Characters

Some of these characters are "special":

'\n'

'\t'

These are still single (individual) characters:

the ***escape sequence*** characters.

ASCII Table

(Characters and Decimal codes)

33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	_
48	0	64	@	80	P	96	`	112	p		

Some Famous Characters

And there is one special character that
is especially special to C strings:

The null terminator character:

`'\0'`

That is an escaped zero.
It’s in ASCII position zero.

It is the value **`0`** (not the character zero, **`'0'`**)

If you output it to screen nothing will appear.

Some Famous Characters

Table 3 Character Literals	
`'y'`	The character y
`'0'`	The character for the digit `0`. In the ASCII code, `'0'` has the value 48.
`' '`	The space character
`'\n'`	The newline character
`'\t'`	The tab character
`'\0'`	The null terminator of a string
`"y"`	**Error:** Not a char value

The Null Terminator Character and C Strings

The null character is special to C strings because it is always the last character in them:

`"CAT"` is really this sequence of characters:

`'C' 'A' 'T' '\0'`

The null terminator character indicates the end of the C string

The Null Terminator Character and C Strings

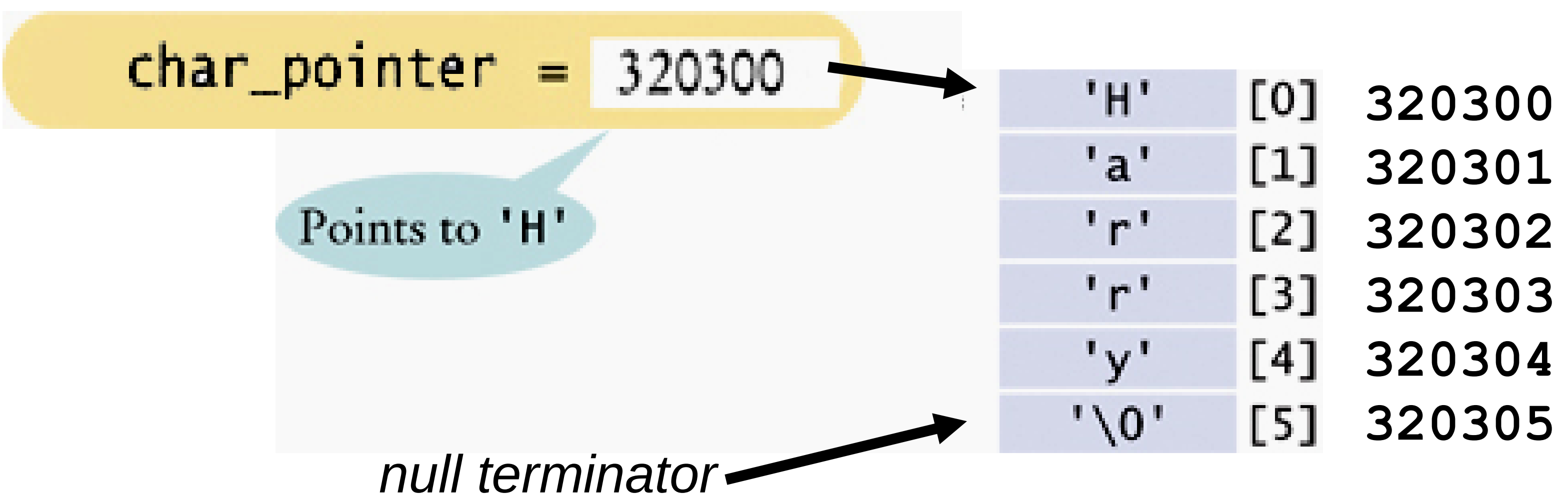
The literal C string **`"CAT"`** is actually an array of ***four*** **`char`**s stored somewhere in the computer.

In the C programming language, literal strings are always stored as character arrays.

Character Arrays as Storage for C Strings

As with all arrays, a string literal can be assigned to a pointer variable that points to the initial character in the array:

```
char* char_pointer = "Harry";
      // Points to 'H'
```



Using the Null Terminator Character

Functions that operate on C strings rely on this terminator.

The strlen function returns the length of a C string.

```
int strlen(const char s[])
{
    int i = 0;
    // Count characters before
    // the null terminator
    while (s[i] != '\0') {
        i++;
    }
    return i;
}
```

Using the Null Terminator Character

The call `strlen("Harry")` returns 5.

The null terminator character is not counted
as part of the "length" of the C string – but it's there.

Character Arrays

Literal C strings are considered constant.

You are not allowed to modify its characters.

char *message = “Hello”;

message[0] = ‘M’; → Illegal

Character Arrays

If you want to modify the characters in a C string, define a character array to hold the characters instead.

For example:

```
// An array of 6 characters
char char_array[] = "Harry";
```

Isn't something missing?

Character Arrays

The compiler counts the characters in the string that is used for initializing the array, including the null terminator.

```
char char_array[] = "Harry";
```

(6)

Character Arrays

You can modify the characters in the array:

```
char char_array[] = "Harry";
char_array[0] = 'L';
```

Example : String assignment

- Outside of declaration statements, assignment of a string variable must be done by the **strcpy** function.
- strcpy (string copy) function is defined in <string.h> header file.

```
char string[20];    // Array declaration

string = "ABCDE";   // Compiler error

strcpy(string, "ABCDE");    // Correct method
```

Error :
Incompatible types in assignment of const char [7] to char [20]

Example: Inputting an Array of Strings

- Program reads 3 names into an array of strings.

```
#include <stdio.h>
#define  N  3      // Number of persons

int main()
{
   int i;
   char  name[N][20]; // Two-dimensional char array

   for (i=0; i < N; i++)
   {
     printf("Enter name of %d. person : ", i+1);
     scanf("%s", name[i]);
     // scanf can read a name that does not contain any spaces
   }

   for (i=0; i < N; i++)
      printf("%s \n", name[i]);
} // end of main
```

Example: Copying an Array of Strings into another Array

- Program copies 3 names from an array of strings into another array.

```c
#include <stdio.h>
#include <string.h>
#define  N  3    // Number of persons

int main() {
  int i;
  char  list1[N][20] = {"AAAA", "BBBB", "CCCC"};
  char  list2[N][20];

  for (i=0; i < N; i++)
  {
      strcpy(list2[i], list1[i]);
      printf("%s \n", list2[i]);
  }
} // end of main
```

Program Output

```
AAAA
BBBB
CCCC
```

Example: ASCII arithmetic

- Program adds an integer value to a char variable to obtain another char value.

```c
#include <stdio.h>

int main()
{
   char x = 'a';      // ascii code 97
   char y = x + 1;    // ascii code 98

   printf("%c  %d \n", x, x);
   printf("%c  %d \n", y, y);
}
```

Program Output

```
a  97

b  98
```

Character Handling Library

- Functions in `<ctype.h>`
- Character handling library
  - Includes functions to perform useful tests and manipulations of character data
  - Each function receives **only one character** as argument (or its ASCII decimal code)

Character Handling Library <ctype.h>

C Function Prototype	Description	Returned value
int **isdigit**(int c);	Returns true if c is a digit and false otherwise.	Zero means false, nonzero means true.
int **isalpha**(int c);	Returns true if c is a letter and false otherwise.	Zero means false, nonzero means true.
int **isalnum**(int c);	Returns true if c is a digit or a letter and false otherwise.	Zero means false, nonzero means true.
int **islower**(int c);	Returns true if c is a lowercase letter and false otherwise.	Zero means false, nonzero means true.
int **isupper**(int c);	Returns true if c is an uppercase letter; false otherwise.	Zero means false, nonzero means true.
int **tolower**(int c);	If c is an uppercase letter, tolower returns c as a lowercase letter. Otherwise, tolower returns the argument unchanged.	Lowercase letter
int **toupper**(int c);	If c is a lowercase letter, toupper returns c as an uppercase letter. Otherwise, toupper returns the argument unchanged.	Uppercase letter

Example: isdigit() function

- Program calls the built-in **isdigit** function to test a char value.

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

int main()
{
  if (isdigit('7'))
      printf("7 is a digit \n");
  else
      printf("7 is a not a digit \n");
}
```

Program Output

```
7 is a digit
```

Example: toupper() function

- Program calls the built-in **toupper** function to convert the char values from lowercase to uppercase.

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

int main()
{
  char letter = 'd';
  printf("%c \n", toupper(letter) );

  printf("%c \n", toupper('z') );
}
```

Program Output

```
D
Z
```

Example: Converting a word to upper case

- Program uses a loop to convert all letters in a word one-by-one to upper case.

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

int main()
{
  char word[] = "Apple";
  int i;

  for (i=0; i < strlen(word); i++)
     word[i] = toupper( word[i] );

printf("%s \n", word);
}
```

Program Output

```
APPLE
```

sprintf (Integer to Ascii)

- Program calls the **sprintf** function to convert an integer value to ASCII string.

```
#include <stdio.h>
int main()
{
   int i = 1234;
   char num[5]; // variable to hold result as a string
   sprintf(num, "%d", i); // prints i to a string, not to screen
   printf( "%d  %s \n", i , num);
}
```

Program Output

```
1234   1234
```

Standard Input/Output Library Functions

- Functions in `<stdio.h>`
- Used to manipulate character and string data

C Function prototype	Function description
int **getchar**( );	Inputs the next character from the standard input and returns it as an integer.
char * **gets**(char *s);	Inputs characters from the standard input into the array s until a newline or end-of-file character is encountered. A terminating null character is appended to the array. (Spaces are also read.)
int **putchar**(int c);	Prints the character stored in c.
int **puts**(const char *s);	Prints the string s followed by a newline character.

Example : getchar()

- **getchar()** function reads one character from keyboard.
- The returned value can be assigned to a char or integer variable.

```
#include <stdio.h>

int main()
{
   char x;

   printf("Enter one character : ");
   x = getchar();
   printf("%d %c \n", x, x);
}
```

The statement
`x = getchar();`

is equivalent to the following statement.

`scanf("%c", &x);`

Program Output

```
Enter one character : a
97  a
```

Example : gets()

- The **gets()** function can read **entire sentence** which may contain spaces.
- If we use the scanf() function, program will read only the first word in sentence.

```
#include <stdio.h>
int main() {
  char sentence[50];
  printf("Enter string : ");

  gets(sentence);   // Read from keyboard

  puts("The string entered was:");
  puts(sentence); // Print to screen
}
```

Program Output

```
Enter string:
ABCD  EFGH

The string entered was:
ABCD  EFGH
```

String Manipulation Functions of the String Handling Library

- Functions in `<string.h>`
- String handling library has functions to
  - Manipulate string data
  - Search strings
  - Determine string length

C Function prototype	Function description
int **strlen**(const char *s)	Returns the length of array s. (Null terminator is not counted).
char * **strcpy**(char *s1, const char *s2)	Copies string s2 into array s1. The new value of s1 is returned.
char * **strncpy**(char *s1, const char *s2, int n)	Copies at most **n** characters of string s2 into array s1. The new value of s1 is returned.
char * **strcat**(char *s1, const char *s2)	Appends string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The new value of s1 is returned.
char * **strncat**(char *s1, const char *s2, int n)	Appends at most **n** characters of string s2 to array s1. The first character of s2 overwrites the terminating null character of s1. The new value of s1 is returned.

Example: String length

`int strlen( const char *s );`

- Returns the number of characters (before `NULL`) in string `s`
- `const` means argument S will be **constant** which can not be modified by the strlen function.

```c
/* fig08_38.c
   Using strlen */
#include <stdio.h>
#include <string.h>

int main() {
   // initialize char pointer
   char *  string = "abc defg";

   printf("The length of %s is %d \n", string, strlen(string));
} // end of main
```

Program Output

```
The length of abc defg is 8
```

Example : String Copying

- Program calls the **strcpy** function to copy the x string into the y string.

```
/* fig08_18.c
   Using strcpy */
#include <stdio.h>
#include <string.h>

int main() {
   char x[] = "Happy Birthday"; // initialize char array x
   char y[ 25 ];

   // copy contents of x into y
   strcpy( y, x );

   printf("The string in array x is: %s \n\n", x);
   printf("The string in array y is: %s \n", y);
} // end of main
```

Program Output

```
The string in array x is: Happy Birthday

The string in array y is: Happy Birthday
```

Example : String Concatenating

- Program calls the **strcat** function to concatanate (append) the s2 string at the end of s1 string.

```
/* Using strcat */
#include <stdio.h>
#include <string.h>
int main() {
   char s1[ 20 ] = "Happy "; // initialize char array s1
   char s2[] = "New Year ";  // initialize char array s2

   printf( "s1 = %s \n s2 = %s \n\n", s1, s2 );

   // concatenate s2 to s1
   strcat( s1, s2 );

   printf( "s1 = %s \n", s1);
} // end of main
```

Program Output

```
s1 = Happy
s2 = New Year

s1 = Happy New Year
```

String Comparison

- Comparing strings
  - Computer compares numeric ASCII codes of characters in string

```
int strcmp( const char *s1, const char *s2 );
```

  - Compares string `s1` to `s2`
  - Returns a negative number if `s1 < s2`
  - Returns zero if `s1 == s2`
  - Returns a positive number if `s1 > s2`

```
int strncmp( const char *s1, const char *s2, int  n );
```

  - Compares up to `n` characters of string `s1` to `s2`

Example : Comparing two strings

- Program calls the **strcmp** functions to compare two strings, and displays the results (0, -1, 1).

```c
/* Using strcmp  */
#include <stdio.h>
#include <string.h>
int main() {
   const char *s1 = "Apple";  // initialize char pointer
   const char *s2 = "Apple";  // initialize char pointer
   const char *s3 = "Grape";  // initialize char pointer
   printf("%s%s\n%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
          "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
          "strcmp(s1, s2) = ", strcmp( s1, s2 ),
          "strcmp(s1, s3) = ", strcmp( s1, s3 ),
          "strcmp(s3, s1) = ", strcmp( s3, s1 ) );
} // end of main
```

Program Output

```
s1 = Apple
s2 = Apple
s3 = Grape

strcmp(s1, s2) =  0
strcmp(s1, s3) = -1
strcmp(s3, s1) =  1
```

ALPHABETICAL COMPARISION RESULTS

s1 (Apple) is equal to s2 (Apple) , result is 0

s1 (Apple) is less than s3 (Grape) , result is -1

s3 (Grape) is bigger than s1 (Apple) , result is 1

Search Functions of the String Handling Library

- Other functions in `<string.h>`

C Function prototype	Function description
char ***strchr**(const char *s, int c);	Locates the first occurrence of character c in string s. If c is found, a pointer to c in s is returned. Otherwise, a NULL pointer is returned.
char ***strstr**(const char *s1, const char *s2);	Locates the first occurrence in string s1 of string s2. If the string is found, a pointer to the string in s1 is returned. Otherwise, a NULL pointer is returned.

Example : strchr()

(Searching char in string)

- Program calls the **strchr** function to search a character in a string.

```
/* fig08_23.c
   Using strchr */
#include <stdio.h>
#include <string.h>

int main()
{
   const char *string = "This is a test"; // initialize char pointer
   char searched = 'a'; // initialize the variable searched

   // if character1 was found in string
   if ( strchr( string, searched ) != NULL )
      printf( "%c was found \n", searched );
   else
      printf( "%c was not found \n", searched);

} // end of main
```

Program Output

```
a was found.
```

Example : strstr()

(Searching string in another string)

- Program calls the **strstr** function to search a string in another string.

```
/* fig08_28.c
   Using strstr */
#include <stdio.h>
#include <string.h>
int main()
{
   const char *string1 = "abcdefabcdef"; // string to search in
   const char *string2 = "def";          // string to search for

   printf( "string1 = %s \n string2 = %s \n", string1, string2);

   if (strstr( string1, string2 ) != NULL )
      printf( "%s was found in string1 \n", string2);
} // end of main
```

Program Output

```
string1 = abcdefabcdef
string2 = def

def was found in string1
```

Array of String Pointers

- Arrays can contain pointers
- `char *` : Each element of `Days` array is a pointer to strings
- The strings are not actually stored in the array `Days`, only pointers to the strings are stored
- Strings are pointers to the first character of each word

```
char * Days[7] = { "Monday", "Tuesday",
                   "Wednesday", "Thursday",
                   "Friday", "Saturday",
                   "Sunday"
                 };
```

Two-dimensional Array of Strings

- Alternative method (as matrix of chars)
- The Days array has a fixed size (7 elements).
- Also, each element of array has a fixed size (10 elements).

```
char Days[7][10] = { "Monday", "Tuesday",
                      "Wednesday", "Thursday",
                      "Friday", "Saturday",
                      "Sunday"
                   };
```

- Displaying all days on screen:

```
for (i=0; i < 7; i++)
    printf("%s \n", Days[i]);
```

Screen Output

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

String-Conversion Functions of the General Utilities Library

Function prototype	Function description
`double strtod(const char *nPtr, char **endPtr);`	Converts the string `nptr` to double.
`long strtol(const char *nPtr, char **endPtr, int base);`	Converts the string `nptr` to long.
`unsigned long strtoul(const char *nPtr, char **endPtr, int base);`	Converts the string `nptr` to unsigned long.

Function `strtod` (1 of 2)

- Function **`strtod`** (Figure 8.6) converts a sequence of characters representing a floating-point value to double.
- The function returns `0` if it's unable to convert any portion of its first argument to double.
- The function receives two arguments—a string (`char *`) and a pointer to a string (`char * *`).
- The string argument contains the character sequence to be converted to double—any whitespace characters at the beginning of the string are ignored.

Function `strtod` (2 of 2)

- The function uses the `char **` argument to modify a `char *` in the calling function (`string ptr`) so that it points to the **location of the first character after the converted portion of the string** or to the entire string if no portion can be converted.
- `d = strtod(string, & stringPtr);`

  indicates that `d` is assigned the double value converted from `string`, and `stringptr` is assigned the location of the first character after the converted value (`51.2`) in `string`.

Using Function `strtod`

```c
// Fig. 8.6: fig08_06.c
// Using function strtod
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   const char *string = "51.2% are admitted"; // initialize string
   char *stringPtr; // create char pointer

   double d = strtod(string, &stringPtr);

   printf("The string \"%s\" is converted to the\n", string);
   printf("double value %.2f and the string \"%s\"\n", d, stringPtr);
}
```

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

Function `strtol` (1 of 4)

- Function **`strtol`** (Figure 8.7) converts to `long int` a sequence of characters representing an integer.
- The function returns `0` if it’s unable to convert any portion of its first argument to `long int`.
- The function receives three arguments—a string (`char *`), a pointer to a string and an integer.
- The string argument contains the character sequence to be converted to `double`—any whitespace characters at the beginning of the string are ignored.

Function `strtol` (2 of 4)

- The function uses the `char **` argument to modify a `char *` in the calling function (`remainderptr`) so that it points to the location of the first character after the converted portion of the string or to the entire string if no portion can be converted.
- The integer specifies the base of the value being converted. The line

```
x = strtol(string, & remainderPtr, 0);
```

indicates that x is assigned the long value converted from string.

Function `strtol` (3 of 4)

- The second argument, `remainderptr`, is assigned the remainder of string after the conversion.
- Using `NULL` for the second argument causes the **remainder of the string to be ignored**.
- The third argument, 0, indicates that the value to be converted can be in octal (base 8), decimal (base 10) or hexadecimal (base 16) format. The base can be specified as 0 or any value between 2 and 36.
  - 0 means strtol automatically decides between base 8, 10, and 16.
- Numeric representations of integers from base 11 to base 36 use the characters A–Z to represent the values 10 to 35.

Function `strtol` (4 of 4)

- When the base argument is set to 0, strtol determines the base automatically based on the format of the input string:
  - If the string starts with "0x" or "0X", it is interpreted as hexadecimal (base 16).
  - If the string starts with "0", it is interpreted as octal (base 8).
  - Otherwise, it is interpreted as decimal (base 10).
- The function returns 0 if it’s unable to convert any portion of its first argument to a `long int` value.

Using Function `strtol`

```c
// Fig. 8.7: fig08_07.c
// Using function strtol
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   const char *string = "-1234567abc"; // initialize string pointer
   char *remainderPtr; // create char pointer

   long x = strtol(string, &remainderPtr, 0);

   printf("%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
      "The original string is ", string,
      "The converted value is ", x,
      "The remainder of the original string is ",
      remainderPtr,
      "The converted value plus 567 is ", x + 567);
}
```

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Using Function `strtol`

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

int main() {
    char *hexStr = "0x1A";
    char *octStr = "075";
    char *decStr = "42";
    char *endptr;

    long hexValue = strtol(hexStr, &endptr, 0);
    long octValue = strtol(octStr, &endptr, 0);
    long decValue = strtol(decStr, &endptr, 0);

    printf("Hexadecimal: %ld\n", hexValue); // Output: 26
    printf("Octal: %ld\n", octValue);       // Output: 61
    printf("Decimal: %ld\n", decValue);     // Output: 42

    return 0;
}
```

C String Functions

Table 4 C String Functions

In this table, s and t are character arrays; n is an integer.	
Function	**Description**
`strlen(s)`	Returns the length of s.
`strcpy(t, s)`	Copies the characters from s into t.
`strncpy(t, s, n)`	Copies at most n characters from s into t.
`strcat(t, s)`	Appends the characters from s after the end of the characters in t.
`strncat(t, s, n)`	Appends at most n characters from s after the end of the characters in t.
`strcmp(s, t)`	Returns 0 if s and t have the same contents, a negative integer if s comes before t in lexicographic order, a positive integer otherwise.