1BEIT105/205 - Programming in C

Module-3

Chapter 4. Arrays and Strings: ***Single-Dimension Arrays, Generating a Pointer to an Array, Passing Single-Dimension Arrays to Functions, Strings, Two-Dimensional Arrays, Multidimensional Arrays, Array Initialization, Variable - Length Arrays. (Pg. 95 – 118)***

Chapter 5. Pointers: ***What Are Pointers? Pointer Variables, The Pointer Operators, Pointer Expressions, Pointers and Arrays, Multiple Indirection, Initializing Pointers. (Pg. 119 – 146)***

Textbook 1: Chapter 4, 5

Chapter 4. Arrays and Strings

Single-Dimension Arrays, Generating a Pointer to an Array, Passing Single-Dimension Arrays to Functions, Strings, Two-Dimensional Arrays, Multidimensional Arrays, Array Initialization, Variable - Length Arrays. (Pg. 95 – 118)

- An ***array is a collection of variables of the same type that are referred to through a common name***. A ***specific element*** in an array is ***accessed by an index***.
- In C, ***all arrays consist of contiguous memory locations***.
  - The ***lowest address*** corresponds to the ***first element*** and
  - the ***highest address*** to the ***last element***.
- Arrays can have from one to several dimensions. The most common array is the string, which is simply an array of characters terminated by a null.

----------**********----------

Single-Dimension Arrays

General Form	*Example*
type var_name[size];	***double balance[100];***

- Like other variables, arrays must be explicitly declared so that the compiler can allocate space for them in memory. Here, "***type"*** declares the ***base type of the array***, which is the type of each element in the array, and size defines how many elements the array will hold.
- For example, to declare a ***100-element*** array called balance of type ***double.***
- In C89, the size of an array must be specified using a constant expression. Thus, the size of an array is fixed at compile time. (C99 allows arrays whose sizes are determined at run time. They are briefly described later in this chapter and examined in detail in Part Two.)
- An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example, ***balance[3] = 12.23;*** assigns element number 3 in balance the value 12.23.
- In C, all arrays have ***"0"*** as the index of their first element. Therefore, when you write ***char p[10];*** you are declaring a character array that has 10 elements, ***p[0]*** through ***p[9].***

```
#include <stdio.h>

int main(void)
{
  int x[100]; /* this declares a 100-integer array*/
  int t;

  /* load x with values 0 through 99 */
  for(t=0; t<100; ++t) x[t] = t;

  /* display contents of x */
  for(t=0; t<100; ++t) printf("%d ", x[t]);

  return 0;
}
```

- The ***amount of storage required to hold an array*** is directly ***related to its type and size***. For a single dimension array, the total size in bytes is computed as shown here:

  total bytes = sizeof(base type) × length of array

- C has no bounds checking on arrays. You could overwrite either end of an array and write into some other variable's data or even into the program's code.
- For example, this code will compile without error, but it is incorrect because the for loop will cause the array count to be overrun.

```
int count[10], i;

/* this causes count to be overrun*/
for(i=0; i<100; i++) count[i] = i;
```

- Single-dimension arrays are essentially lists that are stored in contiguous memory locations in index order. For example, Figure shows how array a appears in memory if it starts at memory location 1000 and is declared as shown here***: char a[7];***

Element	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Address	1000	1001	1002	1003	1004	1005	1006

- You can generate a pointer to the first element of an array by simply specifying the array name, without any index. For example, given int ***sample[10];*** you can generate a pointer to the first element by using the name sample. Thus, the following program fragment assigns p the address of the first element of sample:

```
int *p;
int sample[10];

p = sample;
```

- You can also specify the address of the first element of an array by using the & operator. For example, ***sample*** and ***&sample[0]*** both produce the same results. However, in professionally written C code, you will almost never see ***&sample[0].***

==========

Passing Single-Dimension Arrays to Functions

- You can also specify the address of the first element of an array by using the & operator. For example, ***sample*** and ***&sample[0]*** both produce the same results. However, in professionally written C code, you will almost never see ***&sample[0].***
- In C, you cannot pass an entire array as an argument to a function. You can, however, pass a pointer to an array by specifying the array's name without an index.

```
int main(void)
{
  int i[10];
  func1(i);
  /* . . . */
}
```

- For example, the following program fragment passes the address of ***"i"*** to ***"func1( )":***
- If a function receives a pointer to a single-dimension array, you can declare its formal parameter in one of three ways: as a pointer, as a sized array, or as an unsized array.
- For example, to receive ***"i",*** a function called ***func1( )*** can be declared as

void func1(int *x) {	***void func1(int x[10]) {***	***void func1 (int x[]){***
/ pointer */*	*/* sized array */*	*/* unsized array */*
/* . . . */	*/* . . . */*	*/* . . . */*
}	***}***	***}***

- ***All three declaration methods produce similar results*** because each tells the compiler that an ***integer pointer*** is going to be ***received***.
- The first declaration uses a pointer.
- The second employs the standard array declaration.
- In the final version, a modified version of an array declaration simply specifies that an array of type int of some length is to be received. As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking.

==========

Strings

- The ***most common use*** for the ***one-dimensional array*** is as a ***character string***.
- In C, a ***string is a null-terminated character array***. (A ***null*** is ***zero***.) Thus, a string contains the characters that make up the string followed by a null. The null-terminated string is the only type of string defined by C.
- When declaring a character array that will hold a string, you need to declare it to be one character longer than the largest string that it will hold. For example, ***to declare an array str that can hold a 10-character string***, you would write ***char str[11];*** Specifying 11 for the size makes room for the null at the end of the string.
- When you use a quoted string constant in your program, you are also creating a null-terminated string. A string constant is a list of characters enclosed in double quotes.
- For example: ***'hello there"*** You do not need to add the null to the end of string constants manually— the compiler does this for you automatically.
- C supports a wide range of functions that manipulate strings. The most common are listed here:

Name	Function
strcpy(s1, s2)	**Copies s2 into s1**
strcat(s1, s2)	**Concatenates s2 onto the end of s1**
strlen(s1)	**Returns the length of s1**
strcmp(s1, s2)	**Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2**
strchr(s1, ch)	**Returns a pointer to the first occurrence of ch in s1**
strstr(s1, s2)	**Returns a pointer to the first occurrence of s2 in s1**

- These functions use the standard header ***<string.h>.*** The following program illustrates the use of these string functions:

Program	***Output***
#include <stdio.h> ***#include <string.h>*** ***void main(void)*** ***{*** ***char s1[80], s2[80];*** ***printf("Enter first string: ");*** ***fgets(s1, sizeof(s1), stdin);*** ***printf("Enter second string: ");*** ***fgets(s2, sizeof(s2), stdin);***	*Enter first string:* ***partha*** *Enter second string:* ***sarathy*** *lengths:* ***7 8*** ***partha*** ***sarathy*** ***This is a test.*** ***e is in hello*** ***found hi***

```
 printf("lengths: %d %d\n", strlen(s1), strlen(s2));
 if(!strcmp(s1, s2))
  printf("The strings are equal\n");
 strcat(s1, s2);
 printf ("%s\n", s1);
 strcpy(s1, "This is a test.\n");
 printf(s1);
 if(strchr("hello", 'e')) printf("e is in hello\n");
 if(strstr("hi there", "hi")) printf("found hi");
 return;
}
```

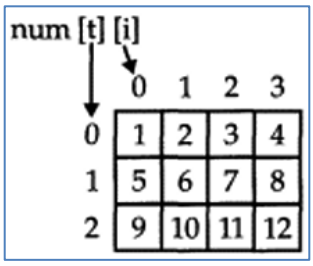
- Remember, ***strcmp( )*** returns false if the strings are equal. Be sure to use the ***logical ! operator*** to reverse the condition, as just shown, if you are testing for equality.

==========

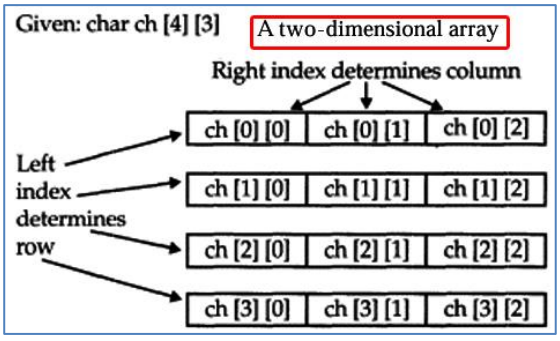
Two-Dimensional Arrays

- C supports multidimensional arrays. The simplest form of the multidimensional array is the two dimensional array.
- To declare a two-dimensional integer array d of size 10,20, you would write int ***d[10][20];***
- Similarly, to ***access point 1,2 of array d***, you would use ***d[1] [2].***
- The following example loads a two-dimensional array with the numbers 1 through 12 and prints them row by row.

Program	***Output***

`#include <stdio.h>`
`int main(void)`
`{`
`  int t, i, num[3][4];`
`  for(t=0; t<3; ++t)`
`  for(i=0; i<4; ++i)`
`  num[t][i] = (t*4)+i+1;`
`  /* now print them out */`
`  for(t=0; t<3; ++t) {`
`  for(i=0; i<4; ++i)`
`  printf(''%3d ", num[t][i]);`
`  printf("\n");`
`  }`
`  return 0;`
`}`
	In this example, ***num[0][0]*** has the value 1, ***num[0][1]*** the value 2, ***num[0][2]*** the value 3, and so on. The value of ***num[2][3]*** will be 12. You can visualize the num array as shown here:
	 

- Two-dimensional arrays are stored in a row-column matrix, where
  - the ***left index*** indicates the ***row*** and
  - the ***right indicates*** the ***column***.
- This means that the rightmost index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory. See Figure for a graphic representation of a two-dimensional array in memory.



- In the case of a two-dimensional array, the following ***formula yields the number of bytes of memory needed to hold it:***

  bytes = size of 1st index × size of 2nd index × sizeof(base type)
- Therefore, assuming
  - integers = 4-byte
  - an integer array with dimensions [10][5] would have
  - bytes allocated = 10 × 5 × 4 = 200 bytes.
- When a ***two-dimensional array is used as an argument to a function, only a pointer to the first element is actually passed***. The parameter receiving a two-dimensional array must define at least the size of the rightmost dimension. The rightmost dimension is needed because the compiler needs to know the length of each row if it is to index the array correctly.
- For example, a function that receives a two-dimensional integer array with dimensions [10][10] can be declared like this:

```
void func1(int x[] [10]) {
    /* . . . */
}
```

- The compiler needs to know the size of the right dimension in order to correctly execute expressions such as ***x[2] [4]*** inside the function. ***If the length of a row is not known, the compiler cannot determine where the next row begins***.
- The following program uses a two-dimensional array to store the numeric grade for each student in a teacher's classes. The program assumes that the teacher has three classes and a maximum of 30 students per class. Notice the way the array grade is accessed by each of the functions.

```
/* A simple student grades database. */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#define CLASSES 3
#define GRADES 30
int grade[CLASSES] [GRADES];
void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);

int main(void)
{
  char ch, str[80];
  for(;;)
   do {
     printf("(E)nter grades\n");
     printf("(R)eport grades\n");
     printf(" (Q)uit\n");
     gets(str);
     ch = toupper(*str);
   } while(ch!='E' && ch!='R' && ch!='Q');

switch(ch) {
    case 'E':
      enter_grades();
      break;
    case 'R':
      disp_grades(grade);
```

```
      break;
    case 'Q':
      exit (0);
   }
  }
  return 0;
}

/* Enter the student's grades. */
void enter_grades(void)
{
  int t, i;
  for(t=0; t<CLASSES; t++) {
    printf("Class # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      grade[t][i] = get_grade(i);
  }
}

/* Read a grade. */
int get_grade(int num)
{
  char s[80];
  printf("Enter grade for student # %d:\n", num+1);
  gets(s);
  return(atoi(s));
}

/* Display grades. */
void disp_grades(int g[][GRADES])
{
  int t, i;
  for(t=0; t<CLASSES; ++t) {
    printf("Class # %d:\n", t+1);
    for(i=0; i<GRADES; ++i)
      printf("Student #%d is %d\n", i+1, g[t][i]);
  }
}
```

==========

Arrays of Strings

- To ***create an array of strings, use a two-dimensional character array***. The size of the
  - ***left dimension*** determines the ***number of strings***
  - the size of the ***right dimension*** specifies the ***maximum length of each string***.
- The following declares an array of 30 strings, each with a maximum length of 79 characters:
  char str_array[30][80];
- It is easy to access an individual string

Specify only the left index.	*Equivalent*
gets(str_array[2]);	***gets(&str_array[2][0]);***

- To understand better how string arrays work, study the following short program, which uses a string array as the basis for a very simple text editor.

```
/* A very simple text editor. */
#include <stdio.h>

#define MAX 100
#define LEN 80

char text[MAX][LEN];

int main(void)
{
  register int t, i, j;

  printf("Enter an empty line to quit.\n");

  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break;
  } /* quit on blank line */

  for(i=0; i<t; i++) {
    for(j=0; text[i][j]; j++)
      putchar(text[i][j]);
    putchar('\n');
  }
  return 0;
}
```

- This program inputs lines of text until a blank line is entered. Then it redisplays each line one character at a time.

==========

Multidimensional Arrays

- C allows arrays of more than two dimensions.

General form	Example
type *name[Size1][Size2][Size3] . . .[SizeN];*	int *m[4][3][6][5];* (Equivalent to) *void func1(int d[][3][6][5]) {* */* . . . */* *}*

- The of a multidimensional array declaration is Arrays of more than three dimensions are not often used because of the amount of memory they require.
- A four-dimensional character array with dimensions ***10,6,9,4*** requires ***10 * 6 *9 *4*** = ***2,160 bytes***. If the ***array held 2-byte integers***, ***4,320 bytes*** would be needed. If the array held ***doubles*** (***assuming 8 bytes per double***), ***17,280*** bytes would be required.
- The ***storage required increases exponentially with the number of dimensions***.
- For example, if a ***fifth dimension of size 10*** was added to the preceding array, then ***172,800 bytes*** would be required.
- ***In multidimensional arrays, it takes the computer time to compute each index. This means that accessing an element in a multidimensional array can be slower than accessing an element in a single-dimension array.***

Indexing Pointers

- Pointers and arrays are closely related. As you know, an array name without an index is a pointer to the first element in the array.

Example

char p[10];	***p == &p[0]*** - evaluates to true because the address of the first element of an array is the same as the address of the array.

- An array name without an index generates a pointer. Conversely, a pointer can be indexed as if it were declared to be an array. For example, consider this program fragment:

```
int *p, i[10];
p = i;
p[5] = 100;  /* assign using index */
*(p+5) = 100; /* assign using pointer arithmetic */
```

- Both assignment statements place the value ***100*** in the sixth element of ***"i".*** The first statement indexes p; the second uses pointer arithmetic. Either way, the result is the same.
- This same concept also applies to arrays of two or more dimensions. For example, assuming that a is a 10-by-10 integer array, these two statements are equivalent: ***a = &a[0] [0].***
- Furthermore, the ***0,4 element*** of a may be ***referenced two ways***:
  1. either by array indexing, ***a[0][4],*** or
  2. by the pointer, ****((int *)a+4).***
- Similarly, element 1,2 is
  1. either ***a[1][2]***
  2. ****((int *)a+12).***
- In general, for any two-dimensional array:
  a[j][k] is equivalent to ****((base type *)a+(j*row length) +k)***
- The cast of the pointer to the array into a pointer of its base type is necessary in order for the pointer arithmetic to operate properly.
- ***Pointers*** are sometimes ***used to access arrays*** because pointer arithmetic is often ***faster than array indexing.***
- A two-dimensional array can be reduced to a pointer to an array of one-dimensional arrays. Therefore, using a separate pointer variable is one easy way to use pointers to access elements within a row of a two-dimensional array.
- The following function illustrates this technique. It will print the contents of the specified row for the global integer array num.

```
int num[10][10];
/* . . . */
void pr_row(int j)
{
  int *p, t;
  p = (int *) &num[j][0]; /*get address of first
                            element in row j */
  for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

- You can generalize this routine by making the calling arguments the row, the row length, and a pointer to the first array element.
- Character arrays that hold strings allow a shorthand initialization that takes the form:
  char array_name[size] = "string";
- For example, this code fragment initializes ***"str"*** to the phrase ***"I like C":***
  char str[9] = "I like C";
- This is the same as writing ***char str[9] = {'I', ' ', 'l', 'i', 'k', 'e',' ', 'C', '\0'};*** Because strings end with a null, you must make sure that the array you declare is long enough to include the null. This is why ***"str"*** is nine characters long even though "I like C" is only eight. When you use the string constant, the compiler automatically supplies the null terminator.
- Multidimensional arrays are initialized the same as single-dimension ones. For example, the following initializes ***"sqrs"*** with the numbers 1 through 10 and their squares.
  int sqrs[10] [2] = { 1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36, 7, 49, 8, 64, 9, 81, 10, 100 };

- When initializing a multidimensional array, you may add braces around the initializers for each dimension. This is called ***subaggregate*** grouping.
- For example, here is another way to write the preceding declaration:
  int sqrs[10] [2] = { {1, 1}, {2, 4}, {3, 9},{4, 16}, {5, 25}, {6, 36}, {7, 49}, {8, 64}, {9, 81}, {10, 100} };
- When using ***subaggregate*** grouping, if you ***don't supply enough initializers*** for a given group, the ***remaining members will be set to zero, automatically***.

==========

Unsized Array Initializations

- Imagine that you are using array initialization to build a table of error messages, as shown here:
  char e1[12] = "Read error\n";
  char e2[13] = ''Write error\n";
  char e3[18] = "Cannot open file\n";
- It is ***tedious to count the characters in each message manually to determine the correct array dimension***. You can let the compiler automatically calculate the dimensions of the arrays.
- If, in ***an array initialization statement***, the ***size of the array is not specified***, the ***compiler automatically creates an array big*** enough to hold all the initializers present. This is called an "***unsized array"***.
- Using this approach, the message table becomes
  char e1[] = "Read error\n";
  char e2[] = "Write error\n";
  char e3[] = "Cannot open file\n";

- Given these initializations, this statement
  printf("%s has length %d\n", e2, sizeof e2)
  will print ***"Write error has length 13".***
- ***"Unsized array initialization"*** allows you to ***change any of the messages without fear of using incorrect array dimensions***.
- ***"Unsized array initializations"*** are not restricted to one-dimensional arrays. For multidimensional arrays, you must specify all but the leftmost dimension. In this way, you can build tables of varying lengths, and the compiler automatically allocates enough storage for them.
- For example, the declaration of sqrs as an unsized array is shown here:
  int sqrs[] [2] = { {1, 1), {2, 4}, {3, 9}, {4, 16}, {5, 25}, {6, 36}, {7, 49}, {8, 64}, {9, 81}, {10, 100} };
- The ***advantage*** of this declaration over the sized version is that ***you may lengthen*** or ***shorten the table without changing the array dimensions***.

==========

Variable - Length Arrays

- In ***C89 array dimensions must be declared using constant expressions***. Thus, in C89 the ***size of an array is fixed at compile time***.
- This is ***not the case for C99***, which adds a ***powerful new feature*** to arrays: ***variable length***.
- In C99, you can ***declare an array whose dimensions are specified by any valid expression***, including those whose ***value is known only at run time***. This is called a ***variable-length array***.
- However, only local arrays (that is, those with block scope or prototype scope) can be of variable length. Here is an example of a variable-length array……

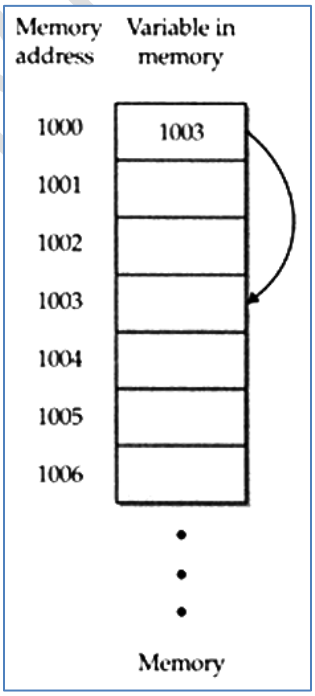
```
void f(int dim)
{
  char str[dim]; /*a variable-length character array*/
  /* . . . */
}
```

- Here, the size of ***"str"*** is determined by the value passed to ***f( )*** in ***dim***. Thus, each call to ***f( )*** can result in ***"str"*** being created with a different length.
- One ***major reason for the addition of variable-length arrays to C99*** is to ***support numeric processing***. Of course, it is a feature that has ***widespread applicability***. But ***remember, variable-length arrays are not supported by C89.***

Chapter 5. Pointers

- Use of pointers is crucial to successful C programming (several reasons).
  - Provide ***functions can modify their calling arguments***.
  - ***Support dynamic allocation***.
  - Can ***improve the efficiency of certain routines***.
  - ***Support for dynamic data structures***, such as ***binary trees*** and ***linked lists.***
- Pointers are one of the strongest but also one of the most dangerous features in C like - program to crash - causing bugs that are very difficult to find.

What Are Pointers?



- A ***pointer is a variable that holds a memory address***. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to point to the second. Figure illustrates this situation.

Pointer Variables

- A pointer declaration consists of a base type, an ***" * ",*** and the variable name.

General form	Example
type *name;	***int *count; float *sum;***

  where ***type*** is the base type of the pointer and may be any valid type. The ***name*** of the pointer variable is specified by name.
- Technically, any type of pointer can point anywhere in memory.
- All pointer operations are done relative to the pointer's base type.
- For example, when you declare a pointer to be of type int *, the compiler assumes that any address that it holds points to an integer— whether it actually does or not. Therefore, when you declare a pointer, you must make sure that its type is compatible with the type of object to which you want to point.

The Pointer Operators

- There are two pointer operators: ***** and ***&***. The & is a unary operator that returns the memory address of its operand. For example, ***m = &count;*** places into ***"m"*** the memory address of the variable ***"count".***

Assume that the variable ***"count"*** uses ***memory location 2000*** to store its value. Also assume that ***count =100;***	***m = &count*** Here **m= 2000**	***m = *count*** Here **m= 100**

Pointer Expressions (Topics covered: Pointer Assignments - Pointer Conversions - Pointer Arithmetic - Pointer Comparisons)

- In general, ***expressions involving pointers conform to the same rules as other expressions***. This section examines a few special aspects of pointer expressions, such as ***assignments***, ***conversions***, and ***arithmetic***.

Pointer Assignments

- You can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straightforward. For example:

```
#include <stdio.h>
int main(void)
{
  int x = 99;
  int *p1, *p2;
  p1 = &x;
  p2 = p1;
  /* print the value of x twice */
  printf(''Values at p1 and p2: %d %d\n", *p1, *p2);
  /* print the address of x twice */
  printf("Addresses pointed to by p1 and p2: %p %p", p1, p2);
  return 0;
}
```

- ***"p1"*** and ***"p2"*** both point to x. Thus, both ***"p1"*** and ***"p2"*** refer to the same object. Sample output from the program, which confirms this, is shown here.

  Values at p1 and p2: 99 99

  Addresses pointed to by p1 and p2: 0063FDF0 0063FDF0

- Notice that the addresses are displayed by using the ***%p printf( )*** format specifier, which causes ***printf( )*** to display an address in the format used by the host computer.

Pointer Conversions

- ***One type of pointer can be converted into another type of pointe***r.
- There are two general categories of conversion: those that involve ***void **** pointers, and those that don't.
- In C, it is permissible to assign
  - a ***void **** pointer to any other type of pointer.
  - any other type of pointer to a ***void **** pointer.
- A ***void **** pointer is called a ***generic pointer***. The ***void **** pointer is used to specify a pointer whose base type is unknown. The ***void * type*** allows a function to specify a parameter that is capable of receiving any type of pointer argument without reporting a type mismatch.
- It is ***also used to refer to raw memory*** (such as that returned by the ***malloc( )*** function) when the semantics of that memory are not known.
- No explicit cast is required to convert to or from a ***void **** pointer.
- Except for ***void ****, all other pointer conversions must be performed by using an explicit cast. However, the ***conversion of one type of pointer into another type may create undefined behavior***. For example, consider the following program that attempts to assign the value of x to y, through the pointer p. This program compiles without error, but does not produce the desired result.

```
#include <stdio.h>
int main(void)
{
  double x = 100.1, y;
  int *p;

  /* The next statement causes p (which is an
     integer pointer) to point to a double. */
  p = (int *) &x;

  /* The next statement does not operate as expected. */
  y = *p; /* attempt to assign y the value x through p */

  /* The following statement won't output 100.1. */
  printf("The (incorrect) value of x is: %f", y);

  return 0;
}
```

- Notice that an ***explicit cast*** is used when assigning the address of x (which is implicitly a double * pointer) to ***"p",*** which is an ***int * pointer***. While this cast is correct, it does not cause the program to act as intended (at least not in most environments).
- To understand the problem, assume ***4-byte int's*** and ***8-byte doubles***. Because ***"p"*** is ***declared*** as an ***integer pointer***, only ***4 bytes*** of information will be transferred to ***"y"*** by this assignment statement, ***y = *p;*** not the ***8 bytes*** that make up a ***double***.
- Thus, even though ***"p"*** is a valid pointer, the fact that it points to a "***double"*** does not change the fact that operations on it expect "***int"*** values. Thus, the use to which ***"p"*** is put is invalid.

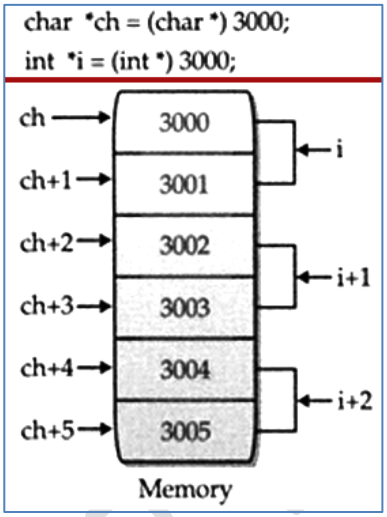
Points to remember

- Pointer operations are performed relative to the base type of the pointer. While it is technically permissible for a pointer to point to some other type of object, the pointer will still ''think" that it is pointing to an object of its base type. Thus, ***pointer operations are governed by the type of the pointe***r, ***not the type of the object being pointed to***.
- You ***can convert an integer into a pointer*** or ***a pointer into an integer***. However, ***you must use an explicit cast***, and the result of such a conversion is implementation defined and may result in undefined behavior.

Pointer Arithmetic

- There are ***only two arithmetic operations that you can use on pointers***:
  1. ***addition***
  2. ***subtraction***.
- To understand what occurs in pointer arithmetic, let ***p1*** be an integer pointer with a ***current value of 2000***. Also, assume ***int's are 2 bytes long***.
- After the expression ***p1++; "p1"*** contains ***2002,*** not ***2001.*** The reason for this is that each time ***"p1"*** is incremented, it will point to the next integer. The same is true of decrements.
- For example, assuming that ***"p1"*** has the value ***2000***, the expression ***p1--;*** causes ***"p1"*** to have the value ***1998.***
- Generalizing from the preceding example, the ***following rules*** govern pointer arithmetic.
  - Each time a pointer is incremented, it points to the memory location of the next element of its base type.
  - Each time it is decremented, it points to the location of the previous element.
- When applied to char pointers, this will appear as ''normal" arithmetic because a ***char object is always 1 byte long no matter what the environment***.
- All other ***pointers will increase or decrease by the length of the data type they point to***. This approach ensures that a pointer is always pointing to an appropriate element of its base type. The ***following figure*** illustrates this concept.

- You are not limited to the increment and decrement operators. For example, you may add or subtract integers to or from pointers. The expression ***p1 = p1 + 12;*** makes **p1** point to the ***12th element of p1's type*** beyond the one it currently points to.
- Besides addition and subtraction of a pointer and an integer, ***only one other arithmetic operation is allowed***:
- You can subtract one pointer from another in order to find the number of objects of their base type that separate the two.
- All other arithmetic operations are prohibited
  - you ***cannot multiply*** or ***divide pointers***;
  - you ***cannot*** add ***two pointers***;
  - you ***cannot apply*** the ***bitwise operators to them***;
  - and you ***cannot add*** or ***subtract type float*** or ***double*** to ***or from pointers***.



Pointer Comparisons

- You can ***compare two pointers in a relational expression***. For ***instance***, given ***two pointers "p"*** and ***"q",*** the following statement is perfectly valid:
  if(p < q) printf("p points to lower memory than q\n");
- Generally, ***pointer comparisons are useful only when two pointers point to a common object, such as an array***.
- As an ***example***, a set of ***stack functions are developed that store and retrieve integer values***. As most readers will know, a stack is a list that ***uses first-in***, ***last-out*** accessing. It is often compared to a stack of plates on a table—the first one set down is the last one to be used.
- ***Stacks*** are used ***frequently in compilers***, ***interpreters***, ***spreadsheets***, and other system-related software. To create a stack, you need two functions: ***push( )*** and ***pop( ).*** The ***push( )*** function places values on the stack, and ***pop( )*** takes them off. These routines are shown here with a simple ***main( )*** function to drive them. The program puts the values you enter into the stack. If you enter "***0"***, a value is popped from the stack. To stop the program, enter "***–1"***.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 50

void push(int i);
int pop(void);

int *tos, *p1, stack[SIZE];

int main(void)
{
  int value;
  tos = stack;  // tos points to the top of stack
  p1 = stack;   // initialize p1

  do {
    printf("Enter value (-1 to quit): ");
    scanf("%d", &value);

    if (value > 0)
      push(value);
    else if (value == 0)
```

```
        printf("Value on top is %d\n", pop());

    } while (value != -1);

    return 0;
}

void push(int i)
{
    p1++;
    if (p1 == (tos + SIZE)) {
        printf("Stack Overflow.\n");
        exit(1);
    }
    *p1 = i;
}

int pop(void)
{
    if (p1 == tos) {
        printf("Stack Underflow.\n");
        exit(1);
    }
    p1--;
    return *(p1 + 1);
}
```

- You can see that memory for the stack is provided by the ***array stack***. The pointer ***"p1"*** is set to point to the ***first element in stack***. The ***"p1"*** variable accesses the stack. The variable ***tos*** holds the ***memory address of the top of the stack***. It is ***used to prevent stack overflows and underflows***.
- Once the stack has been initialized, ***push( )*** and ***pop( )*** can be used. Both the ***push( )*** and ***pop( )*** functions perform a relational test on the pointer ***"p1"*** to detect limit errors. In ***push( ), "p1"*** is tested against the end of the stack by adding ***SIZE*** (the size of the stack) to ***tos***.
- This prevents an overflow. In ***pop( ), "p1"*** is checked against tos to be sure that a stack underflow has not occurred. In pop( ), the parentheses are necessary in the return statement. Without them, the statement would look like this, return ****p1+1;*** which would return the value at location p1 plus one, not the value of the location p1+1.

Pointers and Arrays

- There is a close relationship between pointers and arrays. Consider this program fragment:
  char str[80], *p1;
  p1 = str;
- Here, ***"p1"*** has been set to the address of the first array element in ***"str"***. To access the fifth element in ***str***, you could write ***str[4]*** or ****(p1+4).*** Both statements will return the fifth element.
- Remember, arrays start at ***"0".*** To access the fifth element, you must use ***4*** to index ***"str"***. You also add 4 to the pointer ***"p1"*** to access the fifth element because ***"p1"*** currently points to the first element of ***"str"***.
- C provides ***two methods*** of ***accessing array elements***:
  - **pointer arithmetic** and
  - **array indexing**.

- Although the ***standard array-indexing notation is sometimes easier to understand***, ***pointer arithmetic can be faster***. Since speed is often a consideration in programming, C programmers often ***use pointers to access array elements***.
- These ***two versions*** of ***putstr( )—*** one with array indexing and one with pointers— illustrate how you can use pointers in place of array indexing. The ***putstr( )*** function writes a string to the standard output device one character at a time.

```
/* Index s as an array. */
void putstr(char *s)
{
  register int t;

  for(t=0; s[t]; ++t) putchar(s[t]);
}
/* Access s as a pointer. */
void putstr(char *s)
{
  while(*s) putchar(*s++);
}
```

- Most professional C programmers would find the second version easier to read and understand. Depending upon the compiler, it might also be more efficient. In fact, the pointer version is the way routines of this sort are commonly written in C.

Arrays of Pointers

- ***Pointers*** can be ***arrayed*** like any other data type.
- The declaration for an int pointer array of size 10 is : ***int *x[10];***
- To assign the address of an integer variable called var to the third element of the pointer array, write ***x[2] = &var;***
- To find the value of var, write ****x[2]***
- If you ***want to pass an array of pointers into a function***, you can ***use the same method that you use to pass other arrays***: Simply ***call*** the function with the array name ***without any subscripts***. For example, a function that can receive array x looks like this:

```
void display_array(int *q[])
{
  int t;
  for(t=0; t<10; t++)
    printf("%d ", *q[t]);
}
```

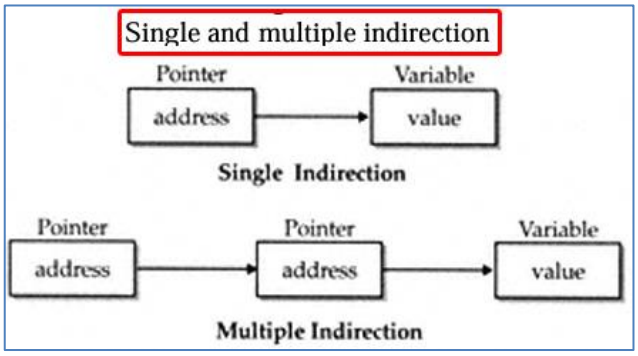
- Remember, ***"q"*** is ***not a pointer to integers***, but rather a ***pointer to an array of pointers to integers***. Therefore, you need to declare the parameter ***"q"*** as an array of integer pointers, as just shown. You cannot declare ***"q"*** simply as an integer pointer because that is not what it is.
- ***Pointer arrays are often used to hold pointers to strings***. For example, you can create a function that outputs an error message given its index, as shown here:

```
void syntax_error(int num)
{
  static char *err[] = {
    "Cannot Open File\n",
    "Read Error\n",
    "Write Error\n",
    "Media Failure\n"
  };
  printf("%s", err[num]);
}
```

- The array ***"err"*** holds a pointer to each error string. This works because a string constant used in an ***expression produces a pointer to the string***. The ***printf( )*** function is ***called with a character pointer that points to the error message whose index is passed to the function***.
- For example, if ***"num"*** is passed a ***"2",*** the message ***"Write Error"*** is displayed. As a point of interest, note that the command line argument ***"argv"*** is an array of character pointers.

Multiple Indirection

- You can have a ***pointer*** point to ***another pointer*** that ***points to the target value***. This situation is called ***multiple indirection***, or ***pointers to pointers***.
- Figure helps clarify the concept of multiple indirection.



- As you can see, the value of a normal pointer is the address of the object that contains the desired value.
- In the case of a pointer to a pointer, the ***first pointer contains the address of the second pointer***, ***which points to the object that contains the desired value***.
- ***To access the target value*** indirectly pointed to by a pointer to a pointer, you must ***apply the asterisk operator twice (**)***, as in this example:

```
#include <stdio.h>
int main(void)
{
  int x, *p, **q;
  x = 10;
  p = &x;
  q = &p;
  printf("%d", **q); /*print the value of x*/
  return 0;
}
```

- Here, ***"p"*** is declared as a pointer to an integer and ***"q"*** as a pointer to a pointer to an integer. The call to ***printf( )*** prints the number 10 on the screen.

Initializing Pointers

- After a nonstatic, local pointer is declared but before it has been assigned a value, it contains an unknown value. (Global and static local pointers are automatically initialized to null.) Should you try to use the pointer before giving it a valid value, you will probably crash your program— and possibly your computer's operating system as well— a very nasty type of error!
- A pointer that does not currently point to a valid memory location is given the value null (which is zero). ***Null is used because C guarantees that no object will exist at the null address***. Thus, ***any pointer that is null implies that it points to nothing and should not be used***.

- One way to give a pointer a null value is to assign zero to it. For example, the following initializes p to null. ***char *p = 0;***
- Additionally, many of C's headers, such as ***<stdio.h>*** define the ***macro NULL***, which is a null pointer constant. Therefore, you will often see a pointer assigned null using a statement such as this: ***p = NULL;***
- However, just because a pointer has a null value, it is not necessarily ''safe." The use of null to indicate unused pointers is simply a convention that programmers follow. It is not a rule enforced by the C language.
- For example, the following sequence, although incorrect, will still be compiled without error:

  int *p = 0;
  ****p = 10; /* wrong! */***
- In this case, the assignment through p causes an assignment at 0, which will usually cause a program crash. Because a null pointer is assumed to be unused, you can use the null pointer to make many of your pointer routines easier to code and more efficient.
- For example, you can ***use a null pointer to mark the end of a pointer array***. A routine that accesses that array knows that it has reached the end when it encounters the null value. The ***search( )*** function shown in the following program illustrates this type of approach. Given a list of ***names, search( )*** determines ***whether a specified name is in that list.***

```
#include <stdio.h>
#include <string.h>
int search(char *p[], char *name);
 char *names[] = {
  "Herb", "Rex", "Dennis", "John", NULL};
  /* null pointer constant ends the list */
int main(void)
{
  if(search(names, "Dennis") != 1)
    printf ("Dennis is in list.\n");
  if(search(names, "Bill") == -1)
    printf("Bill not found.\n");
  return 0;
 }
 /* Look up a name. */
int search(char *p[], char *name)
 {
  register int t;
  for(t=0; p[t]; ++t)
    if(!strcmp(p[t], name)) return t;
    return -1; /* not found */
 }
```

- The ***search( )*** function is ***passed two parameters***.
- The first, ***"p",*** is an array of ***char * pointers*** that ***point to strings containing names***.
- The second, name, is a pointer to a string that points to the name being sought.
- The ***search( )*** function searches through the list of pointers, seeking a string that matches the one pointed to by name.
- The for loop inside search( ) runs until either a match is found or a null pointer is encountered. Assuming the end of the array is marked with a null, the condition controlling the loop is false when the end of the array is reached. That is, p[t] will be false when p[t] is null.
- In the example, this occurs when the name Bill is tried, since it is not in the list of names.
- C programmers commonly initialize ***char **** pointers to point to string constants, as the previous example shows.

- To understand why this works, consider the following statement: ***char *p = "hello world";***
- As you can see, ***"p"*** is a pointer, not an array. This raises a question: Where is the string constant "hello world" being held? Since ***"p"*** is not an array, it can't be stored in ***"p"***. The answer to the question is found in the way C compilers handle string constants. The C compiler creates what is called a string table, which stores the string constants used by the program.
- Therefore, the preceding declaration statement places the address of ''hello world", as stored in the string table, into the pointer ***"p"***. Throughout a program, ***"p"*** can be used like any other string.
- For example, the following program is perfectly valid:

```c
#include <stdio.h>
#include <string.h>
char *p = "hello world";
 int main(void)
{
  register int t;

  /* print the string forward and backwards */
  printf(p);
  for(t=strlen(p)-1; t>-1; t--) printf("%c",
      p[t]);
  return 0;
}
```

Output

```
hello worlddlrow olleh
```

Pointers to Functions

- A ***function has a physical location in memory that can be assigned to a pointer***. This address is the entry point of the function and it is the address used when the function is called. ***Once a pointer points to a function, the function can be called through that pointer.***
- ***Function pointers also allow functions to be passed as arguments to other functions.*** You obtain the address of a function by using the function's name without any parentheses or arguments.
- To see how this is done, study the following program, which compares two strings entered by the user. Pay close attention to the declarations of ***check( )*** and the function pointer ***"p"***, inside ***main( ).***
- Let's look closely at this program. First, examine the declaration for ***"p"*** in ***main().*** It is shown here:
  int (*p)(const char *, const char *);
- This declaration tells the compiler that ***"p"*** is a pointer to a function that has two ***const char ** parameters, and returns an int result.

```c
#include <stdio.h>
#include <string.h>
void check(char *a, char *b,
    int (*cmp)(const char *, const char *));
int main(void)
{
  char s1[80], s2[80];
  int (*p)(const char *, const char *);
  p = strcmp; /* assign address strcmp to p */
  printf("Enter two strings.\n");
  gets(s1);
  gets(s2);
  check(s1, s2, p);
  return 0;
 }
 void check(char *a, char *b,
    int (*cmp) (const char *, const char *))
 {
  printf(''Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
```

- The parentheses around ***"p"*** are necessary in order for the compiler to properly interpret this declaration. You must use a similar form when declaring other function pointers, although the return type and parameters of the function may differ.
- Next, examine the ***check( )*** function. It declares ***three parameters***:
    - ***two character pointers***, **"a"** and **"b"**
    - one ***function pointer***, **cmp**.
- Notice that the function pointer is declared using the same format as was ***"p"*** inside ***main( ).*** Thus, ***cmp*** is able to receive a pointer to a function that takes two ***const char ** arguments and returns an int result. Like the declaration for ***"p"***, the parentheses around the ****cmp*** are necessary for the compiler to interpret this statement correctly.
- When the program begins, it assigns ***"p"*** the address of ***strcmp( ),*** the standard string comparison function. Next, it prompts the user for two strings, and then it passes pointers to those strings along with ***"p"*** to ***check( ),*** which compares the strings for equality. Inside ***check( ),*** the expression ***(*cmp)(a, b)*** calls ***strcmp(***), which is pointed to by ***cmp,*** with the arguments ***"a" and "b".*** The parentheses around ****cmp*** are necessary. ***This is one way to call a function through a pointer***.
- A second, simpler syntax, as shown here, can also be used: ***cmp(a, b);***
- Note that you can call ***check( )*** by using ***strcmp( )*** directly, as shown here: ***check(s1, s2, strcmp);*** This ***eliminates the need for an additional pointer variable***, in this case.
- You can get a better idea of the value of function pointers by studying the expanded version of the previous example, shown next.
- In this version, ***check( )*** can be made to check for either alphabetical equality or numeric equality by simply calling it with a different comparison function. When checking for numeric equality, the string ***"0123"*** will compare equal to ***"123",*** even though the strings, themselves, differ.
- In this program, if you enter a string that begins with a digit, ***compvalues()*** is passed to ***check().*** Otherwise, ***strcmp()*** is used. Since ***check()*** calls the function that it is passed, it can use a different comparison function in different cases.
- Two sample program runs are shown here:

  Enter two values or two strings.
  Test
  Test
  Testing strings for equality.

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void check(char *a, char *b,
   int (*cmp)(const char *, const char *));
int compvalues(const char *a, const char *b);
int main(void)
{
 char s1[80], s2[80];
 printf("Enter two values or two strings.\n");
 gets (s1);
 gets(s2);
 if(isdigit(*sl)) {
    printf(''Testing values for equality.\n");
    check(s1, s2, compvalues);
  }
  else {
    printf("Testing strings for equality.\n");
    check(s1, s2, strcmp);
  }
  return 0;
 }
 void check(char *a, char *b,
    int (*cmp)(const char *, const char *))
 {
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
 int compvalues(const char *a, const char *b)
 {
  if(atoi(a)==atoi(b)) return 0;
  else return 1;
}
```

C's Dynamic Allocation Functions

- Sometimes you will want to allocate memory using ***malloc( ),*** but operate on that memory as if it were an array, using array indexing. In essence, you may want to create a dynamically allocated array. Since any pointer can be indexed as if it were an array, this presents no trouble. For example, the following program shows how you can use a dynamically allocated array to hold a one dimensional array— in this case, a string.

```
 /* Allocate space for a string dynamically */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
  char *s;
  register int t;
  s = malloc(80);

  if(!s) {
    printf("Memory request failed.\n");
    exit (1);
  }
  gets(s);
  for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
  free(s);
  return 0;
}
```

- As the program shows, before its first use, ***"s"*** is tested to ensure that the allocation request succeeded and that a valid pointer was returned by ***malloc( ).*** This is absolutely necessary to prevent accidental use of a null pointer. Notice how the pointer s is used in the call to ***gets( )*** and then indexed as an array to print the string backwards.
