ARRAYS

Beyond the four basic data types (char, int, float, double), the C language supports some advanced data constructs. These include arrays, strings, pointers and structures:

Data construct	**Intended to store:**
array	list of same-type values
string	text
pointer	address of another variable
structure	group of mixed-type values

-An array is intended to group together a list of values, all of the same type, under one variable name. It is much easier to code computations on an array than on a list of independently named variables. Say we have some value x1, x2, x3, …. to get the sum we can do either of the below:

```
for (i=0; i<100; i++) /* array coding */
sum=sum+x[i];
```

or

```
sum=x1+x2+x3+... /* without an array */
```

-A structure is intended to group together an assortment of values of different types under a single variable name. Like an array, it makes it easier to code computations.

-A string is intended to group together a series of character symbols under one variable name. It is closely related to an array, but because it is intended only for text data, a number of functions have been crafted to perform text-specific operations on strings.

-A pointer is intended to hold the address of another variable, to provide a "gateway" or path of indirect access to the other variable. It is used most often in passing values between pieces of code (functions).

1. Arrays

An array is a construct used to store a set of values using only one variable name.

Each of the values occupies a cell. Every cell in the array is the same size, meaning it occupies the same amount of memory. The size of each cell is dictated by the data type (char, int, float, double) given in the variable declaration. The number of cells is also given in the variable declaration. Here are some examples:

```
int a[2];       /*2 cells, each cell 4 bytes (32 bits) */
float b[3];          /*3 cells, each cell 4 bytes (32 bits) */
double c[4];         /*4 cells, each cell 8 bytes (64 bits) */
char d[5];      /*5 cells, each cell 1 bytes (8 bits) */
```

Cells are accessed using indices, with syntax similar to that used in the variable declarations. For example:

```
a[0]=5;
b[1]=4.0;
c[2]=14.7;
d[4]='a';
```

The array bounds for an array say anArray[x] is;

0 to x(total number of cells in array) -1.

However, the C language does not check to make sure that program code stays within the array bounds. It is possible to compile and execute code that goes outside the array bounds. For example, if we add the following code to that from above:

```
b[4]=15.9;

printf("%lf\n",b[4]);
```

Even though index 4 is past the array bounds for the variable b, the output of this code still produces the expected output:

15.900000

This works because the four bytes after b[2] can be accessed using the name b[3]: these bytes occupy addresses used by the variable c[0].

However, the C compiler allows a programmer to access them by “going off the end” of the b array. Past the array bounds, accesses simply move ahead the appropriate number of bytes. The next four bytes, at memory addresses, can be accessed using the name b[4]. These operations “work” because although they are accessing memory outside the array bounds, they are still accessing memory used by this program. Therefore, the program compiles and executes with seemingly correct output.

out-of-bounds access may lead to the following problems:

- It is possible for an access outside the array bounds to clobber a variable. This happens when the out-of-bounds array access overwrites a value in another variable. In our example above, half the variable c[0] is clobbered by writing to b[3], and the other half of c[0] is clobbered by writing to b[4].
  Encountering this error, a programmer will observe a variable to suddenly change value. Seemingly, no line of code in the program changes that variable’s value, yet when displayed it has changed.

- It is also possible for an out-of-bounds array access to cause a program to crash. For example, consider adding the following code fragment to those from above:

  ```
  b[33333]=15.9;
  printf("%lf\n",b[33333]);
  ```

This code still compiles, but when executed produces a segmentation fault. This is because the out-of-bounds array access went far beyond the memory used by this program, and perhaps tried to access memory used by another program (or that was otherwise restricted). The operating system will recognize the problem and terminate the program, reporting a segmentation fault.

- A bus error is a type of crash that occurs when a program tries to access a memory address that is physically impossible (or nonexistant). A bus error can also occur on some systems when a program tries to address an “unaligned” address. This means that the program attempts to read multiple bytes that are not aligned with the width of the data bus (for example, on a 32-bit system the data bus is 4 bytes wide). If an error of this type happens, the operating system will recognize the problem and terminate the program, reporting a bus error.

Why does the C compiler allow out-of-bounds array accesses? The question is actually more complicated than it seems. Not all arrays have a fixed, known size. Sometimes a programmer needs to decide on the size of an array after a program has been compiled and is executing. Sometimes a programmer needs an array to change size while a program is executing. These needs make it impossible for the C compiler to know, in all cases, if an array access is out-of-bounds. The answer may not be known until the program is running and reaches the code that accesses the array.

Multidimensional Arrays

Arrays in C can have more than one dimension. But computer memory is all arranged in one dimensional order, as though it were one long street of bytes. The cells in the multidimensional array are listed out, one at a time, in one-dimensional order. For example, consider the following code:

```
int a[3][2];
a[0][1]=7;
a[1][0]=13;
```

There are a total of six cells in this array. They are listed in order by cycling through the range on the rightmost index, incrementing the next index to the left when done. The same procedure works for any number of dimensions. For example, consider the following code for a three-dimensional array:

```
int b[2][3][4];
b[0][2][0]=7;
b[1][0][2]=13;
```

The order of incrementing dimension indices is similar to how base 10 numbers are counted, cycling through the one’s digit before incrementing the ten’s digit, then cycling through the ten’s digit before incrementing the hundred’s digit, and so on.

NOTE: Everything stored in computer memory is somehow stretched out in one-dimensional order. Things not typically viewed as one-dimensional, such as images, video, databases, maps, and three-dimensional models are all actually stored by listing out the data in one-dimensional order.

STRINGS

A string is a specific type of array: it is an array of char, containing a sequence of values where a value of ’\0’ signifies the end of the string. Although the array could be of any size, it is assumed that the valid data in the array starts at the first cell, and ends with the first cell having a value of ’\0’. For example:

```
char d[8];
d[0]=’H’; d[1]=’e’; d[2]=’l’; d[3]=’l’; d[4]=’o’;
d[5]=’\0’; /* ’\0’ indicates the end of string */
```

Although the array has eight cells, only the first six are used. The cell containing the ’\0’ character is not seen during either printing or the scanning of input. It is a *nonprintable* character used to control how text data is processed. Any code working on a string is supposed to stop processing the array when the ’\0’ character is reached. It is assumed that the values in the remaining cells (two in this case) are not used.

The value of the ’\0’ character is zero. ’\0’ is simply another way of saying zero. It is used to specify a specific “type” of zero. For example, it is informative to use a different zero for a whole number (0) than for a real number (0.0) to show that they are representing slightly different information. ’\0’ is a way to say zero for a char representing an ASCII symbol that means end of string.

Zero has another alias called NULL. NULL is usually used to indicate a value of zero for an address, but it is sometimes used for the end of string character. Any of these aliases can be used; they all result in the same bit pattern being placed in the cell, which is all zeros. For example, all of the following lines of code are equally valid and do the exact same thing:

```
d[5]=’\0’;      /* ASCII zero */
d[5]=0;              /* integer zero */
d[5]=(char)NULL;     /* address zero */
```

The basic functions for reading string input from the keyboard and for writing string output to the screen are scanf() and printf(), respectively.

Just as each basic data type has its own % identifier (char uses %c, int uses %i or %d, float uses %f, double uses %lf), a string uses %s. The %s identifier makes it simpler for a programmer to print out a string. To see how it works, Consider the following code to print out the example from above:

```
printf("%c%c%c%c%c\n",d[0],d[1],d[2],d[3],d[4]);
```

This brute force approach is tedious; code must be written to specifically print out each character, and we must know exactly how many are to be printed. The %s identifier tells printf() to assume the variable is an array of char, ending with a value of zero, and to print each byte using the ASCII bit model. Using this identifier, we can rewrite the printf() as follows:

```
printf("%s\n",d);
```

The variable name d is a label for an address rather than for a value. It identifies an address say 400. Given that address, the printf() function will print out bytes until it encounters a value of zero. Thus, it will print out the bytes at addresses say 400–404, and upon seeing a value of zero at address 405, it stops.

Quiz: What if we were to code the following?

```
printf("%s\n",d[0]); /* what does this do? */
```

When reading input using the scanf() function, the & symbol is required in front of the variable name for the basic data types (char, int, float, and double).

However, it is not required for a string. For example:

```
int x;
float f;
char s[6];
scanf("%d",&x);
scanf("%f",&f);
scanf("%s",s);
```

The variable names x and f each refer to the values stored for those variables.

Similarly, s[0] through s[7] refer to values. The syntax &x is used to identify the address of x, assume is 400. The syntax &f identifies the address of f, which is 404,(assuming its the address that immediately follows) and &(s[0]) identifies the address of s[0], which is 408.
The variable name s is simply a shorthand for writing &(s[0]). The nature of the operation involving the %s identifier in scanf() is to store a series of characters, ending it with a value of zero. This is why it must be given an address, where the storing of characters is to start.

Multidimensional Strings

One of the common uses for multidimensional arrays is to store a list of strings.
Since each string is a one-dimensional array, a list of strings requires a two-dimensional array. For example:

```
char n[2][4];
n[0][0]='T'; n[0][1]='o'; n[0][2]='m'; n[0][3]=0;
n[1][0]='S'; n[1][1]='u'; n[1][2]='e'; n[1][3]=0;
```

Each string in the two-dimensional array can be referenced using the address label for the one-dimensional array that stores it. For example, the following code prints out the two strings:

```
printf("%s %s\n",n[0],n[1]);
```

Both n[0] and n[1] identify addresses (400 and 404, respectively). In the case of a multidimensional array, the variable name by itself provides a third alias for the starting address of the entire array. In this example, &(n[0][0]), n[0], and n all refer to the same thing, the address 400.

String Library Functions

There are a handful of calculations that are common to a large number of text processing problems. These calculations include finding the length of a string and comparing the contents of two strings. Because they are so common, the C standard library has evolved to include functions to perform these calculations.

However, there are five functions that cover the most common calculations and operations:

Function	**What it does**
strlen()	count the total characters in the string
strcmp()	compare two strings, determine if identical
strcpy()	copy one string into another string variable
strcat()	append one string to another string
sprintf()	print formatted output into a string variable

Studying the operations at the memory level is a good way to approach many text processing problems and is particularly helpful when tackling problems outside the scope of the string library functions.

1. String Length: strlen()

Suppose we wish to count the number of characters in a string. For example:

String	Length
“Hello”	5
“H.i;”	4
“h e y”	5

This calculation can be performed by calling the strlen() function:

```
int length;
char s[6];
length=strlen(s);
```

using this function has several benefits;

i. It does save us code especially when having to write code for the same calculation a few hundred (or thousand) times, the savings add up.
ii. calling the same function prevents accidentally inserting an error into the calculation. Even experienced programmers can make mistakes, and the benefit of using proven code can save the time it takes to track down bugs.

2. String Compare: strcmp()

Suppose we wish to compare two strings to determine if they are the same.
Further, if they are different, we desire to know which string first reaches an index having a value less than the same index in the other string. For example:

Strings	Comparison (meaning)
"Hello" vs. "Hello"	0 (same)
"Hello" vs. "Hellp"	−1 (first string smaller)
"Hey" vs. "Hallo"	1 (second sting smaller)
"Hillo" vs. "Hi"	1 (second sting smaller)

We can compare two strings using the following code:

```
int i,a;
char s[4],t[4];
a=strcmp(s,t);
```

For our example, since "Sue" is less than "Sun," the comparison result is −1.
This saves us slightly more code than the strlen() function, but like strlen(), its real value lies in repeated use.

3. String Copy: strcpy()

Suppose we wish to copy the contents of a string to a second string variable. The following code accomplishes this task:

```
int i;
char s[4],t[4];
s[0]='S'; s[1]='u'; s[2]='e'; s[3]='\0';
i=0;
while (s[i] != '\0')
{
t[i]=s[i];
i++;
}
t[i]='\0';
```

Alternatively the same calculation can be performed by calling the strcpy() function:

```
int i;
char s[4],t[4];
strcpy(t,s);
```

The source string comes second in the argument list; the destination string comes first.

4. String Concatenate: strcat()

Suppose we wish to append a string (the addendum) to the end of another string (the original). For example:

Original	Addendum	Result
“Hi”	“ there”	“Hi there”

This calculation can be performed by calling the strcat() function:

```
strcat(s,t);
```

The original string comes first in the argument list; the addendum string comes second. The result is placed in the original string variable.

5. String Print: sprintf()

The sprintf() function works just like the printf() function, except that the output “prints” into a string variable. This can be useful for converting numeric data types into ASCII text, or for creating long strings from multiple components.

For example:

```
char a[24];
float f;
int i;
f=3.72;
i=9;
sprintf(a,"Price %f, qty %d",f,i);
printf("%s\n",a);
```

The output of this code is:

Price 3.720000, qty 9

The memory map for this code is revealing; it highlights the difference between text-storage and value-storage of numbers:

Label	**Address**	**Value**		**Label**	**Address**	**Value**
a[0]	400	’P’		a[13]	413	‘0’
a[1]	401	‘r’		a[14]	414	‘,’
a[2]	402	‘i’		a[15]	415	‘ ’
a[3]	403	‘c’		a[16]	416	‘q’
a[4]	404	‘e‘		a[17]	417	‘t’
a[5]	405	‘ ’		a[18]	418	‘y’
a[6]	406	‘3’		a[19]	419	‘ ’
a[7]	407	‘.’		a[20]	420	‘9’
a[8]	408	‘7’		a[21]	421	0
a[9]	409	‘2’		a[22]	422	
a[10]	410	‘0’		a[23]	423	
a[11]	411	‘0’		f	424-427	3.72
a[12]	412	‘0’		i	428-431	9

How many bytes does it take to store each of the numbers as a numeric value (float or int) versus as text?

String Functions Example

The following code demonstrates several of the string functions, together in a complete program:

```
#include <stdio.h>
/* for printf(), scanf() */
#include <string.h>
/* for strlen(), strcmp() */
main()
{
char
look[80],test[80];
printf("Look for: ");
scanf("%s",look);
while (1)
{
printf("Enter a string (0 to quit): ");
scanf("%s",test);
if (strcmp(test,"0") == 0)
break;
if (strlen(test) < strlen(look))
printf("%s is too short for %s\n",test,look);
else if (strcmp(test,look) == 0)
printf("Found one!\n");
else if (strncmp(test,look,3) == 0)
printf("Started the same...\n");
else
printf("Not what we're looking for\n");
}
}
```

Compiling and executing the code, we obtain the following (the text given at each input was selected to demonstrate the various functions):

```
Look for: sun
Enter a string (0 to quit):     s
s is too short for sun
Enter a string (0 to quit): sun
Found one!
Enter a string (0 to quit): sunny
Started the same...
Enter a string (0 to quit): sleet
Not what we're looking for
Enter a string (0 to quit):     0
```

The last function call, strncmp(), is a variant on strcmp(). It takes a third argument indicating how many characters (three in this example) are to be compared.
If no difference is found between the two strings after that many characters have been compared, then the function returns zero. Otherwise, the strncmp() function works exactly the same as the original strcmp() function.

Note: There are similar variants on the strcpy() and strcat() functions, called strncpy() and strncat().

Nonlibrary Problems
Not every text processing problem is easily solved through use of the library so a programmer should not become too reliant upon the string library functions.
For many problems, individual character processing of string data must be coded.
For example, Suppose we want to remove all occurrences of the letter 'a' from a string, compressing characters to fill any created gaps.
For example:

Original	Result
"Saturday"	"Sturdy"

There is no single function within the C standard library that will accomplish this task. Instead, we must write code to process the string at the character level. The following code accomplishes 'a'-removal:

```
int i,j;
char s[6];
s[0]='a'; s[1]='b'; s[2]='a'; s[3]='c'; s[4]=0;
i=0; j=0;
while (s[i] != 0)
{
if (s[i] != 'a')
{
s[j]=s[i];
j++;
}
i++;
}
s[j]=0;
```

A serious programmer should be well-versed in the common string processing functions of the C standard library and should use them when appropriate but also he/she must also be able to process strings at the character level, in order to accomplish the given task.

Studying the problem at the memory level is often a good approach. Writing out one or two examples in memory, especially those involving the trickiest cases, will often facilitate code design and debugging.

Command Line Arguments

A command line argument is anything typed at the shell prompt after the name of the program to execute. Command line arguments are typically used to provide information about how the user wishes to run a program. For example:

```
Myname@myComputer>$ ls –l -t
```

The command line argument -l tells the program ls to provide a long listing in its output. The command line argument -t causes the output to be sorted according to time last modified. This example shows two command line arguments; it is possible to have any number. They must be separated from each other by one or more spaces.

The full function declaration for main;

i. makes a program aware of its command line arguments.
ii. Define’s when executed, into what variables are the values for the command line arguments placed.

The full function declaration for main():

```
int main(int argc, char *argv[])
```

main has two parameters, int argc and char *argv[].These are the only parameters that are allowed in main and are used in the following ways

- The variable argc is the argument count and passes the number of arguments passed at the command line i.e. it stores the number of command line arguments including the name of the program). For the example ls -l -t, this equals 3.
- char * argv is a pointer to a list of command line arguments which may be accessed as if they were a series of strings. The variable argv stores a list of strings, one string per command line argument. argv can be accessed just like a two-dimensional array.

For example, the following code prints out all the command line arguments, one character at a time:

```
int i,j;
for (i=0; i<argc; i++)
{
j=0;
while (argv[i][j] != '\0')
{
printf("argument No. %d = %c\n" i,argv[i][j]);
j++;
}
printf("\n");
}
```

Each string can be accessed through its own address label, so that the code example just given can be simplified as follows:

```
for (i=0; i<argc; i++)
printf("argument no. %d = %s\n",i,argv[i]);
```

Although the variable argv can be accessed as though it were a two-dimensional array, it actually occupies memory a little differently.

1.Write a program that accepts 6 command line arguments and prints them to string.
2.Write a program that accepts 2 names as command line arguments and prints them as a single line, concatenates and prints them in one line.