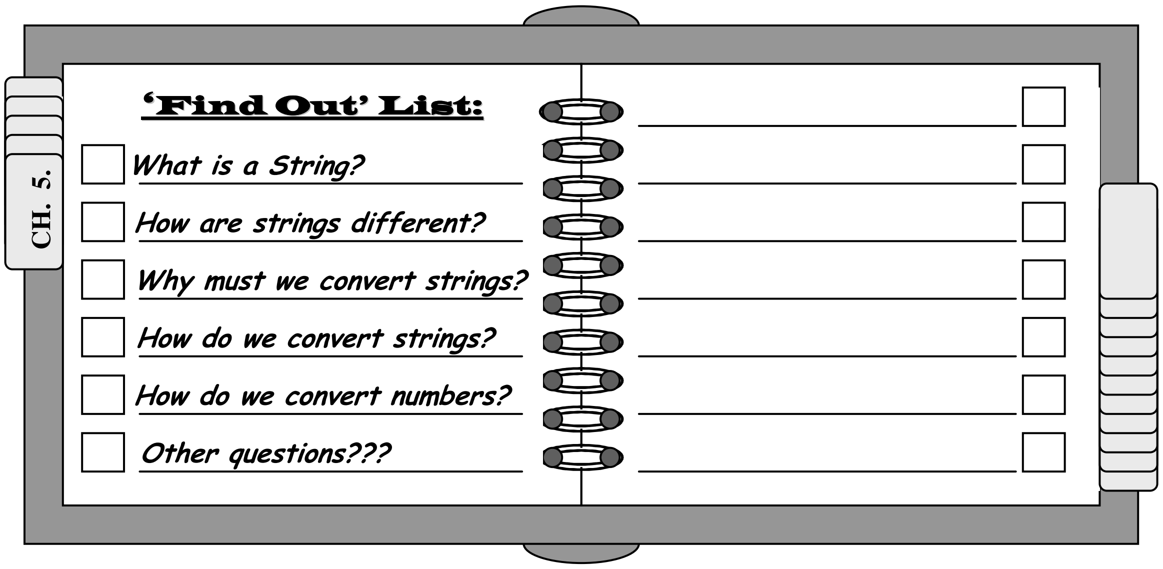
# CHAPTER 5: STRINGS

*"Simplicity of life, even the barest, is not a misery,*
*but the very foundation of refinement"*
*William Morris (1834–96)*



CH. 5.
'Find Out' List:
What is a String?
How are strings different?
Why must we convert strings?
How do we convert strings?
How do we convert numbers?
Other questions???

## Introduction

In some respects, this may seem strange to have a separate chapter on strings since, as we have already mentioned, strings ARE in numeric arrays (i.e., arrays of **char**).Not only are they numeric arrays, they are the simplest of numeric arrays since they only require 1 byte of storage per element.

✓ *What is a String?*

? ***Then why are we discussing strings in a different chapter ???***

Generally speaking, strings are treated differently than the numeric arrays we discussed in the previous chapter. Firstly, we are not really interested in keeping track of string offsets or how many elements comprise a string (although we still have to account for the number of elements when we allocate space). Additionally, because we often must convert 'strings of digits' into their numeric equivalents, and vice versa, strings often require additional manipulation.

With this in mind, we can begin our discussion of strings.

## Strings vs. Numeric Arrays

Like all numeric arrays, strings require a set number of contiguous bytes of storage. In many respects, they are the simplest of arrays, since they require only one byte of storage per element. Therefore, calculating the address of an individual element based on the offset from the base address is relatively simple.

One reason for not discussing strings previously is that when storing strings, we usually need to store an additional element with each array, namely, an end of string marker. To explain, let us start off with a simple example. Assume you wanted to store the string "Hello"

C/C++ Code 5.1.

```
void main()
  { char chararray[5];
    chararray[0] = 'H';
    chararray[1] = 'e';
    chararray[2] = 'l';
    chararray[3] = 'l';
    chararray[4] = 'o';
  }
```

OR, if wished as:

C/C++ Code 5.2.

```
void main()
  { char chararray[5];
    chararray[0] = 72;
    chararray[1] = 101;
    chararray[2] = 108;
    chararray[3] = 108;
    chararray[4] = 111;
  }
```

Both of which would have exactly the same effect.

*Nothing* new here. If we were to assume a base address of 1200 for our array *chararray*, the RAM storage might appear as:

Figure 5.1.

| 1200 | 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 |
|---|---|---|---|---|---|---|---|
| H | e | l | l | o | 10001101 | 10010100 | 01101101 |

Once again, nothing new here. Determining the address of any element in the array is nothing more than the base address (e.g., 1200) plus the offset/subscript (e.g., *chararray*[3] is located at address 1200 + 3 = 1203). If we wished to display the string, we could apply the same code we are accustomed to:

C/C++ Code 5.3.

```
int i;
for (i = 0; i < 5; i++)
   printf("%c", chararray[i]);
```

? ***BUT…. Nothing is Different !!!***

There is only one slight problem. How often do we count the number of characters in a string? For example, how many characters (including spaces and punctuation) were in the previous sentence? (there are 93, by the way)

Because keeping track of the number of characters in a string (as well as the position of each) is very cumbersome, the c programming language, as well as most other programming languages, provides us with a simple method for manipulating strings. If we allow for one additional byte of storage when we declare a character array we can then place a 'special' character (the **NULL** character) at the end of the array. When we do, all that we need to keep track of the base address; the string will start at the base address and continue until we reach the special character.

Suppose we had entered the code:

C/C++ Code 5.4.

```
void main()
{ char chararray[6];
  chararray[0] = 'H';
  chararray[1] = 'e';
  chararray[2] = 'l';
  chararray[3] = 'l';
  chararray[4] = 'o';
  chararray[5] = '\0';
```

where *chararray*[5] (the 6th element in the array) contains the 'special' character we referred to. The character '\0' is actually the null (NULL) character which (as can be determined from the ASCII table) is the number 0 (stored in binary as 00000000). Looking at the relevant section of RAM, we would see:

Figure 5.2.

| 1200 | 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 |
|---|---|---|---|---|---|---|---|
| H | e | l | l | o | \0 | 10010100 | 01101101 |

To print the string chararray, all we need to do is go to the base address (1200) and continue until we encounter the null character. The c code (assuming that we have already included the code given in 5.4.) to do this might look like:

C Code 5.5.

```
int i = 0;
while (chararray[i] != ‘\0’)
   printf(“%c”, chararray[i++]);
```

OR, if we were using pointers:

C Code 5.6.

```
char *i;
i = charaary;
while (*i != ‘\0’)
   printf(“%c”, *i++);
```

Because printing strings is so common, c provides a standard function (in stdio.h) called ***puts*** which essentially contains the above code. To print out the string, we need only issue the command:

C Code 5.7.

```
puts(chararray);
```

There are a number of string functions provided to help deal with strings, including **gets** (from *stdio.h*) which gets a string from the keyboard, and some common utility functions from a file called *string.h* such as **strlen, strcpy**, and **strcmp** (we will discuss some of these a little later).

? ***What if we forget to add the null character to the end of a string???***

All of the string functions discussed above (e.g., ***puts***, ***gets***, ***strcpy***) assume that the null character has been added at the end. If we were to enter the code:

C Code 5.8.

```
void main()
{   char chararray[5];
    chararray[0] = 'H';
    chararray[1] = 'e';
    chararray[2] = 'l';
    chararray[3] = 'l';
    chararray[4] = 'o';
    puts(chararray);
 }
```

we might get the following output:

```
Hello8µ□
```

***Why???***

It depends entirely on what is stored in RAM. To get the above output, the relevant section of RAM would contain:

Figure 5.3.

| | 1200 | 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 | 1208 |
|---|---|---|---|---|---|---|---|---|---|
| | 01001000 | 01100101 | 01101100 | 1101100 | 01101111 | 00111000 | 11100110 | 10000010 | 00000000 |
| <u>Decimal</u>: | 72 | 101 | 108 | 108 | 111 | 56 | 230 | 260 | 0 |
| <u>ASCII</u>: | H | e | l | l | o | 8 | µ | □ | \0 |

Fortunately, many of the common string functions and methods for initializing character arrays automatically include the null character at the end of the string. For example, the declaration:

C Code 5.9.

```
char chararray[6] = "Hello";
```

OR the declaration:

C Code 5.10.

```
char chararray[] = "Hello";
```

***What's the difference ???***

For the first declaration (C code 5.9) we indicated exactly how many contiguous bytes of storage we needed (i.e., 6), including one additional byte for the NULL character. In the second declaration (C code 5.10), we omitted the subscript number. The C compiler will automatically determine how many bytes we need (6 in this case, including the NULL character).

✓ ***How are strings different ?***

## Converting Strings To Numbers

This might initially seem like a strange section to include in our discussion of strings. *Why would we want to convert strings to numbers?* After all, if we wish to enter numbers, we enter numbers.

Well, not quite. When we enter numbers from the keyboard, we are actually entering characters, or more precisely, numeric values which are stored as the data type **char**. When we read-in numbers from as ASCII file, we are actually reading-in characters. To store them in RAM as numeric values (i.e. other than the data type **char**), so that we can perform arithmetic operations on them, we must first convert them to their numeric equivalents. Some of the functions provided by c, such as the ***scanf*** function (which is also found in *stdio.h*), do this for us. However, it is often more efficient to do this ourselves.

✓ ***Why must we convert strings?***

First, let's go over some of the logic involved. Suppose we were trying to convert the string "123" (stored in memory on three contiguous bytes as the numeric values 49, 50, and 51, respectively) to the integer 123 (stored on two-bytes; 1-bit for the sign, 15-bits for the value). If we could identify each of the characters as belonging to the set of digits and determine their position in the character array, we could readily convert the string to its integer value.

***How do we determine if the characters belong to the set of digits???***

That's relatively easy. We know that if the character value is between 48 and 57 (inclusive), or between '0' and '9' (inclusive), which corresponds to the decimal values 48 to 57, we know that they are legal values.

Take the following (5.11.) lines of code:

C Code 5.11.

```
char c[] = “123”;
int legal = 0, i;
for (i = 0; i < 3; i++)
    if ((c[i] < 48) || c[i] > 57))
        legal = 1;
```

All we are doing is checking the contents of each of the locations in the array. If the value of any element in the array is less than 48 (ASCII ‘0’) or greater than 57 (ASCII ‘9’) the character is not legal. If it is not legal, we set the value of the variable legal to 1 (0ne). If we exit the loop and the value of legal is still 0 (zero), the value we initialized it with, the string contains only characters which can be associated with the set of digits. If the value of legal is 1 (one), it contains other characters.

? ***What do the positions of the characters in the array have to do with the value???***

Assume that the string consisted only of the character ‘8’. The corresponding numeric value would also be 8. If the string consisted of the characters ‘76’, however, the corresponding value would be equal to (7 * 10) + 6, or 76. If the string were ‘524’ the numeric equivalent would be (5 * 100) + (2 * 10) + 4 = 524; if it were ‘7838’ the value would be (7 * 1000) + (8 * 100) + (3 * 10) + 8 = 7838.

? ***How do we actually covert???***

Consider the following (5.12.) program (although a little simplistic, it does work):

C Code 5.12.

```
#include <stdio.h>
void main()
{   char nstring[] = “123”;           // the character string we are to convert
    int num = 0,                      // variable num will store the integer equivalent of nstring
        offset = 0;                                             // the offset/index for our array
   while ((nstring[offset] >= '0') && (nstring[offset] <= '9'))  // repeat while we can convert
       num = num * 10 + (nstring[offset++] - '0');               // determine number to date  }
```

The only component which might need a little description are the last two lines of the program. As we already have seen we can keep converting as long as the character we are examining is in the set {‘0’ … ‘9’} (our command while ((nstring[offset] >= '0') && (nstring[offset] <= '9'))). The next line deals with the conversion process we described above. Let’s follow the loop:

Table 5.1.

| *offset* | *nstring*[*offset*] | ***num = num* * 10 + (*nstring*[*offset*++] - '0')** |
|---|---|---|
| 0 | ‘1’ | 1 = 0 * 10 + (‘1’ ( = 49) - ‘0’ ( = 48)) |
| 1 | ‘2’ | 12 = 1 * 10 + (‘2’ ( = 50) - ‘0’ ( = 48)) |
| 2 | ‘3’ | 123 = 12 * 10 + (‘3’ ( = 51) - ‘0’ ( = 48)) |
| 3 | ‘\0’ | * * **loop terminated** * * |

? ***Where does the value of offset get incremented???***

In the command *offset*++ (remember, this is postfix notation; the value of offset is incremented after we use the value *nstring*[*offset*]).

? ***How can offset have the value 3? There are only three characters in the array !!***

Not really. We already noted that the declaration char *nstring*[] = “123”; automatically adds a NULL character at the end of the string.

? ***Why is this program simplistic???***

Because it does not allow for negative integers. Also, it does not allow conversion if the string is preceded by ‘white spaces’ (blanks, carriage returns, and tabs). Consider the code given in code 5.13. This time let’s set up our program as a function which accepts the base address of the character array and returns the integer value represented by the string.

? ***Do we have to duplicate the code given in 5.13. every time we wish to read a numeric value from the keyboard or from an ASCII file?***

We don’t necessarily have to duplicate the code each time, but the conversion must take place. Fortunately, because conversion is so common place, the (almost) identical code is available in the function ***atoi*** which is found in file ***stdlib.h***. Only the commands shown in C code 5.14. need be included in the program.

C Code 5.13.

```
int atoi(const char *stringnum);
void main()
{  int num;
   char *nstring = "123";
   num = atoi(nstring);   }

 int atoi(const char *stringnum)
 {  int n = 0,                                             // The absolute integer value
       sign = 1;                                           // If unsigned, then positive
    while (*stringnum == ' ' || *stringnum == '\n' || *stringnum == '\t')   // skip white spaces
       stringnum++;                                        // move to next position
    if ((*stringnum == '+') || (*stringnum == '-'))        // check if signed
    {   if (*stringnum == '-')                             // if negative
           sign = -1;                                      // then set in the value
       stringnum++;    }                                   // and go the next character
    while ((*stringnum >= '0') && (*stringnum <= '9'))     // Legal value??
    {  n = n * 10 + *stringnum  - '0';                     // determine number to date
        stringnum++;     }                                 // go to next position
    return(sign * n);  }                                   // return the integer
```

C Code 5.14.

```
#include <stdlib.h>
   . . . . .
char *inputstring = “123”;
int num;
num = atoi(inputstring);
```

Notice that *inputstring* is declared as a pointer. Notice also that in case, the value of *num* after the call to **atoi** will be 1234, since the string “1234.56” represents a real number. We could have the string returned as a real number, however, if the call were made to the function **atof** (**A**lpha **TO** **F**loat; also available in file stdlib.h), as in C Code 5.15.

C Code 5.15.

```
#include <stdlib.h>
   . . . . .
char *inputstring = “1234.56”;
float rnum;
rnum = atof(inputstring);
```

✓ ***How do we convert strings ?***

? ***If we wish to store numeric values to an ASCII file, do we first have to convert them to strings???***

YES.

## Converting Numeric Values To Strings

The process may seem a little strange at first, but in fact we have used the process before. Remember how we converted from decimal to binary. For example converting $23_{10}$ to binary:

Calculation 5.1.

| | | |
|---|---|---|
| **23 / 2 = 11** | **23 % 2 = 1** ↑ | $\mathbf{23_{10} = 10111_2}$ |
| **11 / 2 = 5** | **11 % 2 = 1** | |
| **5 / 2 = 2** | **5 % 2 = 1** | |
| **2 / 2 = 1** | **2 % 2 = 0** | *** Remember: we have to collect the bits in reverse order |
| **1 / 2 = 0** | **1 % 2 = 1** | |

If we wished to write a program to convert an integer to binary we would have to store the bits in a string and then reverse the order of the string, just as we did above. Our program might look something like it does in 5.16.:

C Code 5.16.

```
  void main()
 {  int decimal = 23,                    // The integer we wish to convert
        index = 0,                       // one character array index
        offset = 0;                      // another character array index (for swapping)
    char binary[10],                     // the array where we will store the binary number
          tempch;                        // temporary storage the character (for swapping)
    while (decimal > 0)                  // eventually decimal=decimal/2 will yield
decimal=0
     {  binary[index++] = decimal % 2 +'0';  // store the remainder
        decimal = decimal / 2;  }        // get the new quotient
    binary[index--] = '\0';              // set in the null character & decrement the index
    while (index > offset)               // this will our cue stop swapping
     {  tempch = binary[index];          // store the uppermost non-swapped character
        binary[index--] = binary[offset];   // move in the lowermost character & decrement
        binary[offset++] = tempch; }  }     // move in the old uppermost character
```

Once again, let's follow the variables as we go through the loop (See Table 5.2) after we have initialized the integer (**int**) and character (**char**) variables:

Table 5.2.

| *decimal* | *decimal* > 0 ? | *index* | *Binary*[*index*++] = *decimal* % 2 + '0' | *binary* | *decimal*/2 |
|---|---|---|---|---|---|
| 23 | TRUE | 0 | 23 % 2 + '0' = 1 + 48 = 49 = '1' | "1" | 23/2 = 11 |
| 11 | TRUE | 1 | 11 % 2 + '0' = 1 + 48 = 49 = '1' | "11" | 11/2 = 5 |
| 5 | TRUE | 2 | 5 % 2 + '0' = 1 + 48 = 49 = '1' | "111" | 5/2 = 2 |
| 2 | TRUE | 3 | 2 % 2 + '0' = 0 + 48 = 48 = '0' | "1110" | 2/2 = 1 |
| 1 | TRUE | 4 | 1 % 2 + '0' = 1 + 48 = 49 = '1' | "11101" | 1/2 = 0 |
| 0 | **FALSE** | ** | Loop Terminated | | |

After the loop terminates, the next command:

***binary*[*index*--] = '\0';**

puts a null character in *binary*[5] and then decrements the counter (*index* = 4). If we were to look at the string *binary*, we would see:

**"11101\0"**

As we can see, the string is in **reverse** order ($11101_2 = 2^4 + 2^3 + 2^2 + 2^0 = 16 + 8 + 4 + 1 = 29_{10}$ while $10111_2 = 2^4 + 2^2 + 2^1 + 2^0 = 16 + 4 + 2 + 1 = 23_{10}$), so we need to flip it around. Following the variable values as we progress through the loop (Table 5.3):

Table 5.3.

| *index* | *offset* | *index* > *offset?* | *tempch* = *binary*[*index*] | *binary*[*index*--] = *binary*[*offset*] | *binary*[*offset*++] = *tempch* | *binary* |
|---|---|---|---|---|---|---|
| 4 | 0 | TRUE | '1' | '1' | *binary*[0] = '1' | "11101" |
| 3 | 1 | TRUE | '0' | '0' | *binary*[1] = '0' | "10111" |
| 2 | 2 | **FALSE** | *** | Terminate Loop | | |

Converting an integer to a string is performed in exactly the same fashion, except that instead of taking the remainder of the number divided by 2 and converting it into a character, we take the remainder of the number divided by 10 and converting it into a character[1]. We still have to reverse the string once we obtain it, however. Additionally, remember that integers can be signed or unsigned; we have to take that into account also.

Let's rewrite the program again using the function ***itoa*** (**i**nteger **to** **a**lpha), which will take our numeric digit (0, …, 9) and convert it to the numeric equivalent (48, .., 57) as well as function ***reverse***, which will take the ASCII equivalents and store them in reverse order. Lest we get out of practice, let's use pointers (these will continue to gain importance). C Code 5.17. illustrates how this can be done.

[1] This procedure works for any base

C Code 5.17.

```
#include <string.h>                         // we need this for function strlen
 char itoa(int num, char *numstring);       // itoa prototype
 char reverse(char *strg);                  // our reverse function prototype

 void main()
 {  int number = -582;                      // the integer to convert
    char string[11];                        // where we will place the converted integer
    itoa(number, string);                   // call itoa passing the integer and the string
  }

 char reverse(char *strg)                   // this will reverse the order of the string
 {  int offset1,offset2,tempoffset;         // the offsets and temp storage
    for (offset1 = 0, offset2 = strlen(strg) - 1; offset1 < offset2; offset1++, offset2--)
   {   tempoffset = strg[offset1];          // store the lowermost character
       strg[offset1] = strg[offset2];       // move the uppermost to the lowermost
       strg[offset2] = tempoffset;   }      // move the lowermost to the uppermost
 }
 char itoa(int num, char *numstring)        // this will establish the string
 {   char ch, *startstring;                 // temporary storage
     startstring = numstring;               // Store the base address of the string
     int sign = 1;                          // assume the number is unsigned
     if (num < 0)                           // is the number negative?
    {    num = -num;                        // then make it an absolute number
         sign = -1;      }                  // and note the sign
    while (num > 0)                   // now keep determining the quotient
    {   *numstring++ = num % 10 + '0';      // store the remainder
        num /= 10;   }                      // get the new quotient
    if (sign == -1)                         // if it is a negative number
       *numstring++ = '-';                  // add the sign to the end
    *numstring = '\0';                      // include the null character
    reverse(startstring);                   // and reverse the string
 }
```

**? *How would this work???***

In function **main**, we are storing the value –582 in location *number*, requesting 11 bytes of contiguous storage at location *string*, and then passing the numeric value –582 AND the base address of *string* to function *atoi* (which will store the numeric value at address *num* and the address of *string* at location *numstring*). Let's assume that the address of *string* is 7850 and follow what happens to each of the variables in function *atoi*.

The first instructions we encounter:

```
int sign = 1;                  // assume the number is unsigned
if (num < 0)                   // is the number negative?
{    num = -num;               // then make it an absolute number
     sign = -1;      }         // and note the sign
```

simply initializes (integer) location *sign* with the value 1, and then checks to see if it should actually be –1. Since *num* < 0, we change the sign of contents of num (now, +582) and assign the value –1 to location *num*. The loop:

**while** (*num* > 0)

Executes in much the same manner as we saw in table 5.4, namely it converts each digit to its ASCII equivalent, and stores them at address *string* (which we stored at address *numstring*). The only real difference is that instead of storing the remainder of a number divided by 2 (and then adding the value 48 to get the ASCII value), we store the remainder of the number divided by 10 (and then add 48 to get the ASCII value).

The notation might seem a little strange at first, but that is because we are trying to illustrate some of the 'shortcuts' we discussed previously. The command

***numstring++ = num % 10 + '0';**

is really the same as the command ***binary*[*index*++] = *decimal* % 2 +'0';** which we saw in C Code 5.16. Instead of incrementing the array offset (after we store the ASCII equivalent of the remainder, we are incrementing the address of the address (remember, our base address is *string*) by 1 (remember, we are incrementing a character address). Again, we are storing the ASCII equivalent of the remainder. The following statement:

***num* /= 10;**

Simply assigns the quotient of the contents of location *num* to location *num*. Let's track the variables through the loop:

Table 5.4.

| *num* | *num > 0* | *numstring* | **numstring++ = decimal % 10 + '0'* | **numstring* | *num*/10 |
|---|---|---|---|---|---|
| 582 | TRUE | 7850 | 582 % 10 + '0' = 2 + 48 = 50 = '2' | "2" | 582/10 = 58 |
| 58 | TRUE | 7852 | 58 % 10 + '0' = 8 + 48 = 56 = '8' | "28" | 58/10 = 5 |
| 5 | TRUE | 7854 | 5 % 10 + '0' = 5 + 48 = 53 = '5' | "285" | 5/10 = 0 |
| 0 | **FALSE** | ** | Loop Terminated | | |

?

***But the address contained at location numstring is now 7854??? The base address of string (in function main) is 7850!!!***

That is why we stored the base address at location *startstring*.

There are three additional notes to be made here:

1. The numeric value originally passed was negative. That is why we include the statement

   ```
   if (sign == -1)
       *numstring++ = '-';
   ```

   which will add the character '-' to the string and increment the address contained at location *numstring*. If we were to look at location *numstring* in RAM, we would find the string "285-".

2. Strings should contain a NULL character at the end. That is why we include the statement

   ```
   *numstring = '\0';
   ```

   If we were to look at location *numstring* in RAM, we would find the string "285-\0".

3. No matter how we look at it, the string (except for the NULL character) is in reverse order. That is why we pass the base address of the string (which we previously stored in location *startstring*) to function *reverse*.

Function reverse receives as its only input the base address of our string (variable *string* in function **main**). The loop that we apply:

```
for (offset1 = 0, offset2 = strlen(strg) - 1; offset1 < offset2; offset1++, offset2--)
```

1. Initializes *offset1* to 0 (*offset1* will point to the left-end of the unordered string)
2. Initializes *offset2* to the right end of the string. Since the string we pass is "285-\0" (a length of 4, since the NULL character doesn't count in the length) , we set the offset to *strlen*(*strg*) – 1 since we do not wish to change the position of the NULL character.
3. Continues as long as *offset1* is less than *offset2*.
4. Increments *offset1* by 1, and decrements *offset2* by 1, as long as our check (#3) remains TRUE.

The procedures followed correspond directly to what we saw previously. If we were to track the variable values (remember, at this point in time, **str* would be "285-") through the loop, we would see:

Table 5.5.

| *offset1* | *offset2* | *offset1< offset2* | *tempoffset = strg[offset1* | *strg[offset1] = strg[offset2];* | *strg[offset2] = tempoffset;* | **strg* | *offset1* | *offset2* |
|---|---|---|---|---|---|---|---|---|
| 0 | 3 | TRUE | '-' | '2' | '-' | "-852" | 1 | 2 |
| 1 | 2 | TRUE | '8' | '5' | '8' | "-582" | 2 | 1 |
| 2 | 1 | FALSE | ** | Terminated | | | | |

Which is the correct value of the numeric value –582 translated to the ASCII characters "-582".

✓ ***How do we convert numbers ?***

**?** ***Must we ALWAYS duplicate the above code in order to store integers in ASCII format???***

Once again, the c programming language provides the function ***itoa*** (as well as ***atoi***, and ***atof***, and ***atol***, and ***fota***, and, … well conversions between all of the standard data types) in ***stdlib.h***, so we do not always have to write it (See C Code 5.18.: Check the parameters to be passed; they *do* vary slightly from those previously).

C Code 5.18.

```
#include <stdio.h>                // we will print out the value as a string this time
#include <string.h>               // we need this for function strlen
#include <stdlib.h>               // We need this for all of our conversions
void main()
{  int number = -582;             // the integer to convert
   char string[11];               // where we will place the converted integer
   string =  itoa(number);        // call itoa passing the integer and the string
   puts(string);                  // print out the string "-582"
}
```

As we might imagine, the output of our program would be:

```
-582
```

## Summary

Strings are in many respects the simplest of all array types since they take up 1-byte (8-bits) of storage per element and calculating the address of any element is extremely simple (simply add the array offset to the base address). However, because we typically are unconcerned about how many elements are in an array, or what the position of each element is, we have to devise new approaches to dealing with them, most notably, the addition of a NULL character ('/0') at the string. Also, because conversion between strings and numeric variables is so commonly performed, we need to be familiar with the different approaches necessary to convert them.

## Chapter Terminology: Be able to fully describe these terms

file ***stdlib.h***.
function **atof**
function ***atof***
function ***atoi***
function ***atol***
function ***fota***
function **gets**
function ***itoa***
function ***puts***
**NULL** character

## Review Questions

1. Explain why strings are viewed differently from numeric arrays.
2. Explain how strings are stored differently in RAM than are numeric arrays
3. Given:

   **char** *mystring*[5];

   AND we learn that the base address of *mystring* is **8356**. If we were to look in RAM, we would find:

| 8354 | 8355 | 8356 | 8357 | 8358 | 8359 | 8360 | 8361 | 8362 | 8363 | 8364 |
|---|---|---|---|---|---|---|---|---|---|---|
| 01100000 | 00010010 | 01010000 | 01110101 | 01110010 | 01100101 | 00100000 | 01000111 | 01100001 | 11001110 | 01100010 |
| 8365 | 8366 | 8367 | 8368 | 8369 | 8370 | 8371 | 8372 | 8373 | 8374 | 8375 |
| 01100001 | 01100111 | 01100101 | 11111110 | 00000000 | 10110011 | 10110010 | 00010000 | 00000000 | 11001110 | 00100110 |

   If we were to issue the command:

   *puts*(mystring);

   what would the output be??

4. Given:

   **char** *helloarray*[] = “hello”;

   Show EXACTLY how this would be stored in RAM assuming a base address of 12000

5. Explain what the function **atoi** does.
6. Explain what the function **ftoa** does.

✓ ***Other Questions?***

## Review Question Answers (NOTE: CHECKING THE ANSWERS BEFORE YOU HAVE TRIED TO ANSWER THE QUESTIONS DOESN'T HELP YOU AT ALL)

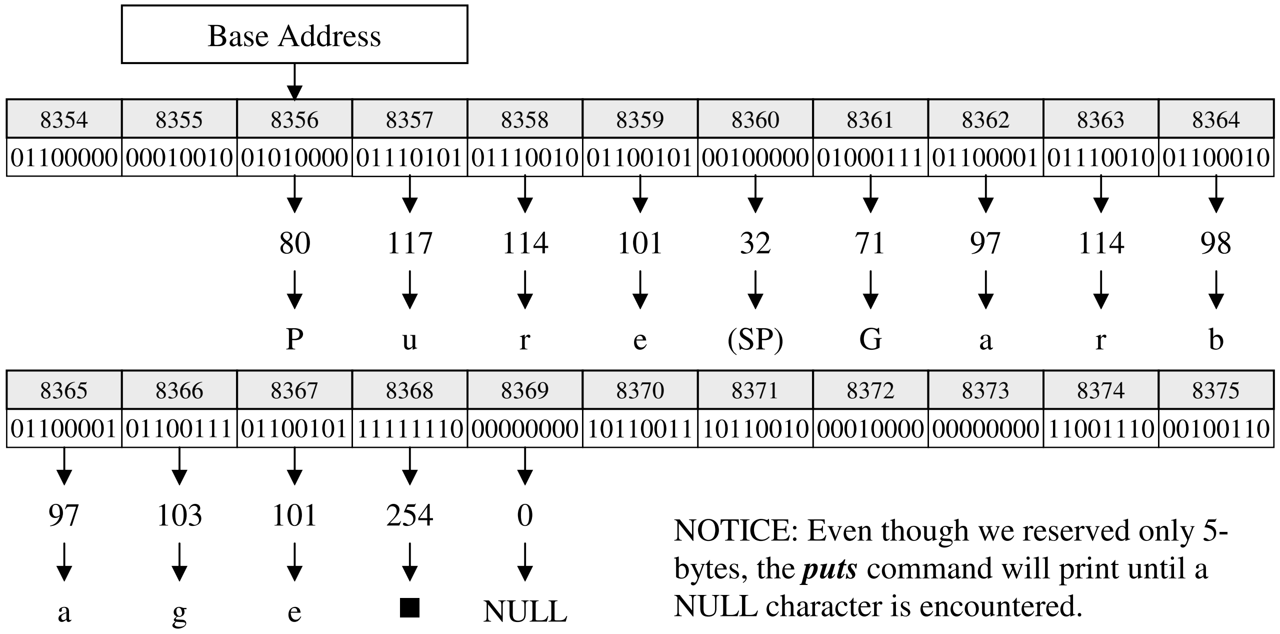
1. Explain why strings are viewed differently from numeric arrays.

   **Strings ARE numeric arrays of type character, but we are not necessarily interested in each offset from the base address**

1. Explain how strings are stored differently in RAM than are numeric arrays

   **Strings require 1 additional byte at the end of the array: the NULL ('/0') character**

3. Given: c**har** *mystring*[5]; with a base address of: **8356**.

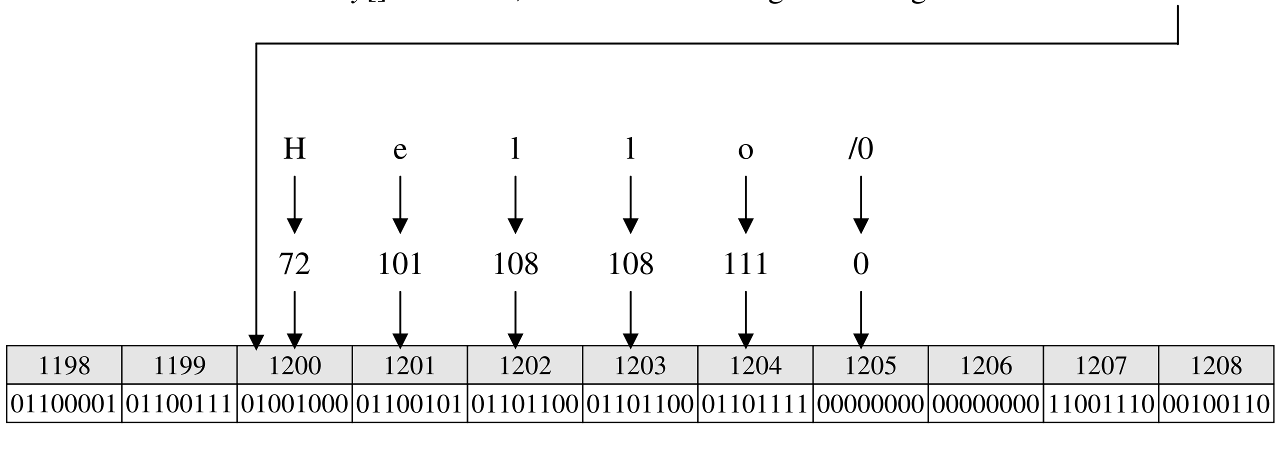


| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 8354 | 8355 | 8356 | 8357 | 8358 | 8359 | 8360 | 8361 | 8362 | 8363 | 8364 |
| 01100000 | 00010010 | 01010000 | 01110101 | 01110010 | 01100101 | 00100000 | 01000111 | 01100001 | 01110010 | 01100010 |
| | | 80 | 117 | 114 | 101 | 32 | 71 | 97 | 114 | 98 |
| | | P | u | r | e | (SP) | G | a | r | b |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 8365 | 8366 | 8367 | 8368 | 8369 | 8370 | 8371 | 8372 | 8373 | 8374 | 8375 |
| 01100001 | 01100111 | 01100101 | 11111110 | 00000000 | 10110011 | 10110010 | 00010000 | 00000000 | 11001110 | 00100110 |
| 97 | 103 | 101 | 254 | 0 | | | | | | |
| a | g | e | ■ | NULL | | | | | | |

NOTICE: Even though we reserved only 5-bytes, the ***puts*** command will print until a NULL character is encountered.

4. Given: **char** *helloarray*[] = "hello"; Show RAM Storage assuming a base address of 1200



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | H | e | l | l | o | /0 | | | |
| | | 72 | 101 | 108 | 108 | 111 | 0 | | | |
| 1198 | 1199 | 1200 | 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 | 1208 |
| 01100001 | 01100111 | 01001000 | 01100101 | 01101100 | 01101100 | 01101111 | 00000000 | 00000000 | 11001110 | 00100110 |

**5.** Explain what the function **atoi** does. **Converts a string to an integer**

**6.** Explain what the function **ftoa** does. **Converts a real number (float) to a string**