Strings

What we have seen so far are arrays of integers and double values. Is it possible to have an array of characters? But of course!

Check code samples (down) below.

```
#include <stdio.h>

int main(){
    char st[5];

    st[0] = 'h';
    st[1] = 'e';
    st[2] = 'l';
    st[3] = 'l';
    st[4] = 'o';

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

    return 0;
}
```

Simply change the data type of the array to `char` from `int`. Seeing the code above will tell us that it is no different from the array of ints. Except that the elements are characters and so they have to enclosed in the single quotes. And the format specifier used in the `printf` is `%c` for character. An array of characters is called a string.

Take a look at the next one (down) below.

```
#include <stdio.h>

int main(){
    char st[] = {'h','e','l','l','o'};

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

    return 0;
}
```

This is similar to initializing an array of integers but this time using character literals. The size of *st* here is 5.

In c, there is a `%s` format specifier. This is used for strings only. Instead of printing the characters one by one in the printf, we can use `%s` and print the string as one entity. This is illustrated (down) below.

```c
#include <stdio.h>

int main(){
    char st1[] = {'h','e','l','l','o'};
    printf("st1: %s\n",st1);

    return 0;
}
```

```c
#include <stdio.h>

int main(){
    char st2[5];

    st2[0] = 'w';
    st2[1] = 'o';
    st2[2] = 'r';
    st2[3] = 'l';
    st2[4] = 'd';
    printf("st2: %s\n",st2);

    return 0;
}
```

Actually, we can just put these 2 in one source file.

```c
#include <stdio.h>

int main(){
    char st1[] = {'h','e','l','l','o'};
    char st2[5];

    st2[0] = 'w';
    st2[1] = 'o';
    st2[2] = 'r';
    st2[3] = 'l';
    st2[4] = 'd';
    printf("st1: %s\nst2: %s\n",st1, st2);

    return 0;
}
```

Compile and run these 3 source files. You should notice some weird characters printed as well.

Try this next one.

```c
#include <stdio.h>

int main(){
    char st1[] = "hello";
    char st2[10];

    printf("Enter a string: ");
    scanf("%s", st2);

    printf("st1: %s\nst2: %s\n",st1, st2);

    return 0;
}
```

Compile and run the code above as well. For st2, enter "world". Notice that the weird unknown characters are no longer printed or displayed with st1 and st2. Take note of the `scanf` as well. Recall that `scanf` requires a memory location where the input has to be stored. The names of arrays (either numerical arrays or strings) hold the memory address of the first element of the array. In short array names are pointers (this is discussed in detail in Functions). For now, suffice it ti say that pointers are types of data that can hold memory addresses only. Hence, the `scanf` didn't fail.

Strings are a special kind of array. The `%s` format specifier in `printf` looks for the string terminator character (a character that symbolizes the end of the string has been reached) so it knows when to stop printing. This string terminator is the NULL character. It is `\0`. Using the `%s` with `scanf` automatically adds `\0` at the end of the string. Declaring the string *st*1 through a string literal (a sequence of characters enclosed in double quotes) also automatically inserts `\0` at the end of the string. As a consequence, the size of *st*1 is 6 instead of 5. The additional size is from the `\0`.

Redoing it as demosntrated (down) below, solves the problem.

```c
#include <stdio.h>

int main(){
    char st1[] = {'h','e','l','l','o','\0'};
    char st2[6];

    st2[0] = 'w';
    st2[1] = 'o';
    st2[2] = 'r';
    st2[3] = 'l';
    st2[4] = 'd';
    st2[5] = '\0';
    printf("st1: %s\nst2: %s\n",st1, st2);

    return 0;
}
```

Notice that we intentionally increased the size of *st*2 to 6 so we can accommodate the string terminator. The size of *st*1 is 6 as well.

In an array of integers, we have to explicitly know the size of the array. With strings, it is possible to scan through the elements one by one and when `'\0'` is encountered, scanning can stop. Also note that the size of the string may 100 for instance, but the actual number of characters present in the string is less than 100.

```
#include <stdio.h>

int main(){
    char st1[20] = "hope";
    int i;

    for(i=0; i<20; i++)
        printf("%c ",st1[i]);

    printf("\n");
    return 0;
}
```

Although the size of *st*1 is 20, “hope” is only 4 characters. What will `printf` display from supposed 5th character all the way to the 20th.

We will use `'\0'` to scan through the string st1. Recall that when strings are taken from standard input or initialized with literal strings, the compiler inserts `'\0'` at the end of the string. So let’s use this information.

```
#include <stdio.h>

int main(){
    char st[20] = "hope";
    int i;

    for(i=0; st[i]!='\0'; i++)
        printf("%c ",st[i]);

    printf("\n");
    return 0;
}
```

What the loop above does is it scans through the elements of the *st* and prints it one by one stopping only when the character of *st* at index i is `'\0'`.

We call i (after the execution of the loop) the length of the string. Since “hope” is 4 letters long, then its **length** is 4. 20 on the other is the size of the *st* - the maximum capacity or maiximum number of characters the string can contain. The last character of the string is found at index $length - 1$.

```c
#include <stdio.h>

int main(){
    char st[20] = "hope";
    int i;

    for(i=0; st[i]!='\0'; i++);

    printf("The length of %s is %i.\n",st,i);

    return 0;
}
```

Some of the string problems we have encountered in flowcharts. If in raptor we used Length_Of, in c there is a function called `length` that accepts a string as an argument and returns the length of the string. It is found in the `string.h` library.

Given a string let's count the occurrences of vowels.

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

int main(){
    char st[20] = "hope";
    int i,len, count;

    printf("Enter a string: ");
    scanf("%s", st);

    len = length(st);

    for(i=0,count=0; i<len; i++){
        if(st[i] == 'a' || st[i] == 'e' || st[i] == 'i' ||
        ↪  st[i] == 'o' || st[i] == 'u')
            count++

    }

    printf("There are %i vowels in  %s\n",count,st);

    return 0;
}
```

This one (down) below checks whether the string is composed of digits only.

```
#include <stdio.h>

int main(){
    char str[100];
    int i, digits=1;

    printf("Enter a string: ");
    scanf("%s",str);

    for(i=0; str[i]!='\0';i++){
        if( !(str[i]>='0' && str[i]<='9') ){
            digits = 0;
            break;
        }
    }

    if(digits)
        printf("%s is composed of digits only!\n",str);
    else
        printf("%s has characters other than
 ↪  digits!\n",str);

    return 0;
}
```

`(str[i]>='0' && str[i]<='9')` checks if `str[i]` is any of the digits from `'0'` to `'9'`. We saw this when we did characters in c. Instead of checking the non-digit characters one by one, and there are so many of them, simply check if the character is a digit and not negate it. Exactly as how we did it with characters in an earlier section.

digit above is used as a flag. Initially, it is assumed that *str* is composed of digits only. But once the solution encounters a non-digit it sets *digit* to false and then breaks from the loop. And prints the appropriate message.

Palindrome problem

We have seen this problem with flowcharts. When a string is spelled backwards and it produces the same string, it is called a palindrome. We discovered from last time that in order for this to solved, two indices have to used. One that starts from the left (index 0) and the other from the right (index $length - 1$). We check whether the characters at these indices are the same character. If they are not, then obviously it is not a palindrome. The loop terminates. And displays the appropriate message. If the characters are the same, we check further if the next characters match as well. This repeats until first half of characters are checked against the second half of characters. Then and only then can we conclude that the string is a palindrome.

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

int main(){
    char str[100];
    //i and j are the 2 indices
    int i,j,len,palindrome = 1; // str is also assumed to
    ↪  be a palindrome

    printf("Enter a string: ");
    scanf("%s",str);
    len = strlen(str);

    for(i=0,j=len-1; i < j; i++, j--){
        if( str[i] != str[j] ){
            palindrome = 0;
            break;
        }
    }

    if(palindrome)
        printf("%s is a palindrome.\n",str);
    else
        printf("%s is not a palindrome\n",str);

    return 0;
}
```

Based on the trace above, when i is now equal to j, the iteration or loop terminates (`i < j`). Notice that when this is the case, we are inspecting the same character. There is, then, no need to check this. This is always the case when the length of the palindrome is odd.

In a similar manner, the trace above terminates when `i < j` evaluates to false. This happens when i and j cross values as show in the image above. This always happens when the length of the palindrome is even.

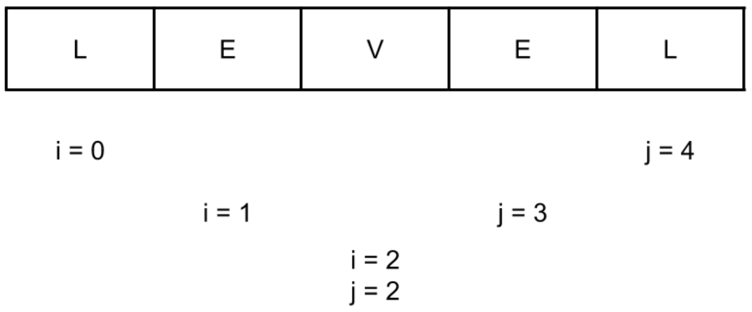

Figure 4: LEVEL Sample Trace

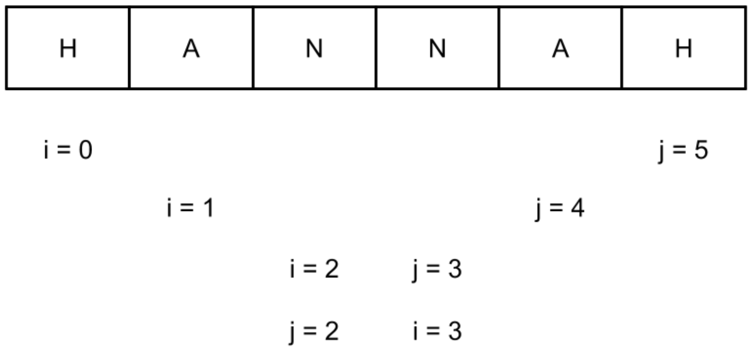

Figure 5: HANNAH Sample Trace

String Comparison

One of the most common string operations is to compare whether two strings are the same string. This is especially helpful with username and password authentications. The username and password in the database must match the username and password provided by the user as inputs.

This is accomplished through character by character comparison. It is not as stright forward as comparing two numbers where the equality check is simply used. It should be also obvious that two strings with different lengths are never the same string.

Check out our solution (down) below.

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

int main(){
    char st1[50], st2[50];
    int len1, len2, i;

    printf("Enter 2 strings: ");
    scanf("%s%s",st1,st2);

    len1 = strlen(st1);
    len2 = strlen(st2);

    if(len1 == len2){
        for(i=0; i<len1; i++)    //we can use either len1 or
        ↪   len1 since they have the same length
            if(st1[i]!=st2[i])  //when the characters in
            ↪   the same position do not match, break
                break;

        if(i < len1)     //this means not all characters of
        ↪   the strings have been exhausted
            printf("They are not the same string!\n");
        else
            printf("They are the same string!\n");
    }
    else // st1 and st2 do not have the same length
        printf("They are not the same string!\n");
    return 0;
}
```

Check this other sample that compares two strings in a different manner.

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

int main(){
    char st1[50], st2[50];
    int len1, len2, i, status=0, len;

    printf("Enter 2 strings: ");
    scanf("%s%s",st1,st2);

    len1 = strlen(st1);
    len2 = strlen(st2);
    len = len1;
    if(len1 < len2)
        len = len2;

    for(i=0; i<len; i++){
        if(st1[i] < st2[i]){
            status = -1;
            break;
        }
        else
        if(st1[i] > st2[i]){
            status = 1;
            break;
        }
    }
    if(status == 0){    //this means status was never
    ↪  updated
        // this means st1 and st2 are the same string
        // but if they are of different lengths
        // the shorter has the same exact characters as
        // the longer string up to index (len - 1).
        if(len1 < len2)
            status = -1;
        else
        if(len2 < len1)
            status = 1;
    }

    if(status == 0)
        printf("%s and %s are the same string.\n",
↪   st1,st2);
    else
    if(status < 0)
        printf("%s comes before %s when they are
↪   alphabetized.\n", st1,st2);
    else
    if(status > 0)
        printf("%s comes after %s when they are
↪   alphabetized.\n", st1,st2);

    return 0;

}
```

The string comparison above doesn't just compare whether two strings are the same. It also identifies which between the two comes first if they were to be alphabetized or arranged in a lexicographic manner. Just note though that what was compared here are their ASCII codes. This means that `'Q'` will definitely come before `'a'`. As a remedy to this problem, perhaps it is best to make sure that both strings are in lowercase or both are in uppercase. Recall this conversion in the Branching section. Perhaps introduce temporary variables to store the inputs in so that we get to keep the original inputs.

String Copy

Compile the code below.

```c
#include <stdio.h>

int main(){
    char st1[50] = "Hello Philippines", st2[50] = "and
    ↪  Hello World!";

    st1 = st2;

    printf("%s %s\n",st1,st2);

    return 0;
}
```

You will get an “unassignable” error. Copying one string to another has to be done character by character. This also means that we have to make sure that source string is shorter in size (capacity and not length of the string) than the destination string.

```c
#include <stdio.h>

int main(){
    char st1[50] = "Hello Philippines"; //source string
    char temp[50];  //destination string
    int i;

    for(i=0; st1[i]!='\0'; i++)
        temp[i] = st1[i];

    temp[i] = '\0'; // never forget the string terminator

    printf("%s is copied from %s.\n",temp,st1);
    return 0;
}
```

Change Case

We already know how to do this with a character. Let’s extend this with a string.

```c
#include <stdio.h>

int main(){
    char st[50];
    int i;

    printf("Enter a string: ");
    scanf("%s",st);

    for(i=0; st[i]!= '\0'; i++){
        if(st[i]>='A' && st[i]<='Z') //only convert
        ↪  uppercase letters to lowercase
            st[i] = st[i] + 'a' - 'A';

    }

    printf("%s\n",st);

    return 0;

}
```

Now, we are ready to redo the solution for string comparison. For this problem, we are making sure that user only enters strings with letters only.

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

int main(){
    char st1[50], st2[50],tmp1[50], tmp2[50];
    int len1, len2, i, status=0, len, valid;

    //makes sure that the only acceptable inputs for st1
    ↪  are those with letters only
    do{
        printf("Enter first string with letters only: ");
        scanf("%s",st1);
        valid = 1;
        for(i=0; st1[i]!='\0'; i++){
            if(!(st1[i]>='A' && st1[i] <='Z' || st1[i]>='A'
            ↪  && st1[i] <='Z')){
                valid = 0;
                break;
            }
        }
    }while(!valid);

    //makes sure that the only acceptable inputs for st2
    ↪  are those with letters only
    do{
        printf("Enter second string with letters only: ");
        scanf("%s",st2);
        valid = 1;
        for(i=0; st2[i]!='\0'; i++){
            if(!(st2[i]>='A' && st2[i] <='Z' || st2[i]>='A'
            ↪  && st2[i] <='Z')){
                valid = 0;
                break;
            }
        }
    }while(!valid);
```

```c
// store st1 in a temporary string tmp1
for(i=0; st1[i]!='\0'; i++)
    tmp1[i] = st1[i];

tmp1[i] = '\0';

// store st2 in a temporary string tmp2
for(i=0; st2[i]!='\0'; i++)
    tmp2[i] = st2[i];

tmp2[i] = '\0';

// makes sure the tmp1 is in lowercase
for(i=0; tmp1[i]!= '\0'; i++){
    if(tmp1[i]>='A' && tmp1[i]<='Z') //only convert
     ↪  uppercase letters to lowercase
        tmp1[i] = tmp1[i] + 'a' - 'A';
}

// makes sure the tmp2 is in lowercase
for(i=0; tmp2[i]!= '\0'; i++){
    if(tmp2[i]>='A' && tmp2[i]<='Z') //only convert
     ↪  uppercase letters to lowercase
        tmp2[i] = tmp2[i] + 'a' - 'A';
}
```

```
    len1 = strlen(st1);
    len2 = strlen(st2);
    len = len1;
    if(len1 < len2)
        len = len2;

    for(i=0; i<len; i++){
        if(tmp1[i] < tmp2[i]){
            status = -1;
            break;
        }
        else
        if(tmp1[i] > tmp2[i]){
            status = 1;
            break;
        }
    }

    if(status == 0){    //this means status was never
    ↪   updated
        // this means st1 and st2 are the same string
        // but if they are of different lengths
        // the shorter has the same exact characters as
        // the longer string up to index (len - 1).
        if(len1 < len2)
            status = -1;
        else
        if(len2 < len1)
            status = 1;
    }

    if(status == 0)
        printf("%s and %s are the same string.\n",
↪   st1,st2);
    else
    if(status < 0)
        printf("%s comes before %s when they are
↪   alphabetized.\n", st1,st2);
    else
    if(status > 0)
        printf("%s comes after %s when they are
↪   alphabetized.\n", st1,st2);

    return 0;
}
```

The new solution above now disregards the case when comparing the characters from the two strings. What we did was to perform the comparison in lowercase case letters.

Looking at the new solution, we realize that we were able to accomplish the solution by decomposing the solution into smaller sub-tasks. These sub-tasks are (1) making sure that the input strings are composed of letters only, (2) store the input strings in temporary variables, (3) convert all the characters of the temporary strings to lowercase letters, and finally (4) perform the comparison task. And this is essentially how we have defined programming at the beginning of this journey!