CSCI 240 Lecture Notes - Part 11

The string data type

The data type called *string* which we have used a little is really an example of a C++ class. We will study more about classes at the end of the course. The point is, it is not really a data type in the same sense as int, double, char, etc...

Using the string class, we did not know how strings were represented (stored) internally. (How much memory do they take up? Do they grow and shrink? Is there a maximum size? etc...)

Thus we did not have to specify and control certain internal details, and this simplified things somewhat, as compared to...

Standard C/C++ strings

The C++ language does **not** have a **string data type**. But there is a way to represent and manipulate sequences of characters (which are usually *called* strings, just as a short way to refer to them).

1. There is *no such thing as a **string data type*** in standard C++.
2. strings are created as *arrays of char*.
3. A **null** character (ASCII 0 or '\0') in the array signals the effective end of the array of char; that is, it marks the end of the string. Characters in the array beyond the **null** are ignored by the functions designed to work on these strings.
4. A set of standard string manipulation functions are provided in the interface defined in <cstring>. (This contains a set of function prototypes to functions in the C++ standard libraries.)

Since a string is an array of char, you can manipulate it with array notation (subscripts).

However, it is often easier to use the string functions - which mostly use an *address-based* notation rather than array notation. They do *not* use a reference notation.

We will look at both approaches.

How do you create a string?

1. Define an array of char.

```
char s[80];
```

Note that we hereby reserve 80 bytes of memory for this string. We are not obligated to use it all, but we better not overfill it or we will be in trouble (because we would overwrite memory after it).

2. Put some chars in it and end it with a null. For now, let's look at the string as an array. So we could do this:

```
s[0] = 'H';     //H as a char
s[1] = 'e';
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = '\0';
```

Note that because chars and ints are interchangeable in C++, so we could also assign the integer 0 to s[5] since that is the ASCII value for the null character.

However, we **cannot do this** to insert the null character:

```
s[5] = '0';     //Char '0' is ASCII 48, not ASCII 0
```

So we have an 80-byte array, and we use only the first 6 bytes.

However, we can now print it using *cout*:

```
cout << s;              //prints: Hello
```

If we had forgotten to put in the terminating null, this *cout* could print something like:

```
Hello. %x6^^* #8, ,d @ b,,.....
```

That is, it would print bytes in memory starting at s, interpreted as chars, until it found (by chance) a null (zero-valued byte).

There are simpler ways to initialize a string:

1. Based on the fact that a string is an array of char, we can do the following:

```
char s[80] = {'H', 'e', 'l', 'l', 'o'};
```

in this case, the rest of the array would contain all 0's.

2. There is a shortcut that is allowed by C++:

```
char s[80] = "Hello";
```

in this case also, the rest of the array would contain all 0's.

3. Looking ahead, there is a string function that will copy one string value into another:

```
char s[80];

strcpy(s, "Hello");
```

in this case, the rest of the array (after the terminating null) would contain whatever was there before the *strcpy()*.

Since a string is an array of chars, we can look at individual chars in the string using array notation.

```
cout << s[2];           //displays: l
```

Suppose we want a function that will return the Ith character of a string. We could write it as follows:

```
char IthChar(char s[], int i)
  {
  return s[i];
  }
```

And we could call it as:

```
cout << IthChar(s, 2);       // displays: l
```

Note how we designed the function:

- it returns a char
- it takes 2 arguments:
  - an array of char (the string)
  - the subscript of the char desired

Note also what would happen if we used:

```
char ch;

ch = IthChar(s, 10);      //or

ch = IthChar(s, j);       //if j has a value > 5
```

It would return some random byte (char) that happened to be in memory at 10 bytes (or j bytes) beyond s - at s[10] or s[j]. But the actual effective string is just "Hello". So we don't know what is in s[10] or s[j].

We could find out, of course, if we *cout*-ed the character returned, but it would be meaningless. It was never assigned a value, so its value is random.

Suppose we want to write a function to change a string to all uppercase.

```
void strUpper( char s[] )
{
int i;            //used as a subscript here

for ( i = 0; s[i] != '\0'; i++ )
  s[i] = toupper(s[i]);
}
```

The ANSI C and C++ String Library functions.

C++ has standard library functions to manipulate strings which are stored as null-terminated arrays of char.

Every vendor who markets a C++ compiler implements these functions, which are then callable from any C++ program.

With C++ standard string functions you have to:

- create memory for each string by declaring a char array
- be sure (one way or the other) that the array of chars has a null terminator
- be sure that you don't overflow the array

Here is a summary of a small subset of the string functions:

The arguments called s1 and s2 are **names of arrays of char**.

A "valid string" is one with a null terminator.

strcpy(s1, s2)
This function will copy s2 into s1. s1 may or may not have a valid string in it. s2 must be a valid string (either a character array with a null terminator OR a string literal). s2 must not be bigger than s1 or array overflow will result.

strcat(s1, s2)
This function concatenates s2 to the end of s1. Both s1 and s2 must be valid strings, but s2 could be a string literal. There must be room in s1 for the entire result.

int strlen(s1)
This function returns the integer length of s1. This is the number of chars in s1 but DOES NOT include the null terminator.

int strcmp(s1,s2)
This function compares the two string arguments alphabetically. It returns:

- a positive value if s1 is alphabetically greater than s2
- a negative value if s1 is alphabetically less than s2
- 0 if s1 is alphabetically equal to s2

For example, suppose you have:

char s1[9], s2[9];

and both have valid strings in them. Suppose you want to concatenate s2 to the end of s1. You could do a test:

```
if (strlen(s1) + strlen(s2) < 9)
  {
  strcat(s1, s2);
  }
else
  {
  cout << "not enough room";
  }
```

More examples will be given in lecture.