C Strings (Arrays vs. Pointers)

---

1. **Strings as arrays:**

   Before the `string` class, the abstract idea of a string was implemented with just an array of characters. For example, here is a string:

   ```
   char label[] = "Single";
   ```

   What this array looks like in memory is the following:

   ```
   -----------------------------
   | S | i | n | g | l | e | \0 |
   -----------------------------
   ```

   where the beginning of the array is at some location in computer memory, for example, location `1000`.

   Note: Don't forget that one character is needed to store the *nul character* (`\0`), which indicates the end of the string.

   A character array can have more characters than the *abstract string* held in it, as below:

   ```
   char label[10] = "Single";
   ```

   giving an array that looks like:

   ```
   -----------------------------------------
   | S | i | n | g | l | e | \0 |   |   |   |
   -----------------------------------------
   ```

   (where 3 array elements are currently unused).

   Since these strings are really just arrays, we can access each character in the array using subscript notation, as in:

   ```
   cout << "Third char is: " << label[2] << endl;
   ```

   which prints out the third character, **n**.

   A disadvantage of creating strings using the character array *syntax* is that you must say ahead of time how many characters the array may hold. For example, in the following array definitions, we state the number of characters (either implicitly or explicitly) to be allocated for the array.

   ```
   char label[] = "Single";  // 7 characters

   char label[10] = "Single";
   ```

   Thus, you must specify the maximum number of characters you will ever need to store in an array. This type of array allocation, where the size of the array is determined at compile-time, is called *static allocation*.

2. **Strings as pointers:**

   Another way of accessing a *contiguous* chunk of memory, instead of with an array, is with a *pointer*.

Since we are talking about *strings*, which are made up of *characters*, we'll be using *pointers to characters*, or rather, `char *`'s.

However, pointers only hold an address, they cannot hold all the characters in a character array. This means that when we use a `char *` to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

Below is how you might use a *character pointer* to keep track of a string.

```
char label[] = "Single";
char label2[10] = "Married";
char *labelPtr;

labelPtr = label;
```

We would have something like the following in memory (e.g., supposing that the array `label` started at memory address `2000`, etc.):

```
label @2000
-----------------------------
| S | i | n | g | l | e | \0 |
-----------------------------

label2 @3000
-----------------------------------------
| M | a | r | r | i | e | d | \0 |   |   |
-----------------------------------------

labelPtr @4000
--------
| 2000 |
--------
```

Note: Since we assigned the pointer the address of an *array of characters*, the pointer must be a *character pointer*--the types must match.

Also, to assign the address of an array to a pointer, we do not use the *address-of* (`&`) operator since the name of an array (like `label`) behaves like the address of that array in this context.

Now, we can use `labelPtr` just like the array name `label`. So, we could access the third character in the string with:

```
cout << "Third char is: " << labelPtr[2] << endl;
```

It's important to remember that the only reason the pointer `labelPtr` allows us to access the `label` array is because we made `labelPtr` point to it. Suppose, we do the following:

```
labelPtr = label2;
```

Now, no longer does the pointer `labelPtr` refer to `label`, but now to `label2` as follows:

```
label2 @3000
-----------------------------------------
| M | a | r | r | i | e | d | \0 |   |   |
-----------------------------------------

labelPtr @4000
--------
```

```
| 3000 |
--------
```

So, now when we subscript using `labelPtr`, we are referring to characters in `label2`. The following:

```
cout << "Third char is: " << labelPtr[2] << endl;
```

prints out **r**, the third character in the `label2` array.

3. **Passing strings:**

   Just as we can pass other kinds of arrays to functions, we can do so with strings.

   Below is the definition of a function that prints a label and a call to that function:

```
void PrintLabel(char the_label[])
{
  cout << "Label: " << the_label << endl;
}

...

int main()
{
  char label[] = "Single";
  ...
  PrintLabel(label);
  ...
```

   Since `label` is a character array, and the function `PrintLabel()` expects a character array, the above makes sense.

   However, if we have a pointer to the character array `label`, as in:

```
char *labelPtr = label;
```

   then we can also pass the pointer to the function, as in:

```
PrintLabel(labelPtr);
```

   The results are the same. *Why??*

   Answer: When we declare an array as the parameter to a function, we really just get a pointer. Plus, arrays are always automatically passed by reference (e.g., a pointer is passed).

   So, `PrintLabel()` could have been written in two ways:

```
void PrintLabel(char the_label[])
{
  cout << "Label: " << the_label << endl;
}

OR

void PrintLabel(char *the_label)
{
  cout << "Label: " << the_label << endl;
}
```

   There is no difference because in both cases the parameter is really a *pointer*.

Note: In C++, there is a difference in the use of brackets (`[]`) when declaring a global, static or local array variable *versus* using this array notation for the parameter of a function.

With a parameter to a function, you always get a *pointer* even if you use array notation. This is true for **all** types of arrays.

---

4. **Dynamically-allocated string:**

   Since sometimes you do not know how big a string is until run-time, you may have to resort to dynamic allocation.

   The following is an example of dynamically-allocating space for a string at run-time:

```
void SomeFunc(int length)
{
  char *str;

  // Don't forget extra char for nul character.
  str = new char[length + 1];

  ...

  // Done with str.
  delete [] str;
  ...
```

   Basically, we've just asked `new` (the allocation operator) to give us back enough space for an array of the desired size. Operator `new` requires the type of elements (here, `char`) and the number of elements needed (given as the *array size* between `[` and `]`).

   Note that unlike static allocation, e.g.:

```
char name[20];
```

   the size can be variable (when using `new` for allocation).

   We keep track of the dynamically-allocated array with a pointer (e.g., the return value of the call to `new` is stored in `str`).We then can use that pointer as we used pointers to statically-allocated arrays above (i.e., we access individual characters with `str[i]`, pass the string to a function, etc.).

   Finally, note that when we are done using the string, we must deallocate it.

Note: Note the use of `[]` when using `delete` to deallocate an array.

Here, we deallocate the string in the same function, but in some cases we might still need it after `SomeFunc()` ends, so we'd deallocate it later. Remember, we must keep a pointer to the beginning of the string because that's what is used to access the string and to deallocate it.

---

5. **Arrays vs. Pointers:**

   As we can now see, there is more than one way to view a *string*. A string might be accessed in a statically-allocated array, it might be accessed via a pointer to a statically-allocated array, or perhaps via a pointer to a dynamically-allocated array.

BU CAS CS - C Strings (Arrays vs. Pointers)