Topic 3

1. Defining and using pointers
2. Arrays and pointers
3. C and C++ strings
4. Dynamic memory allocation
5. Arrays and vectors of pointers
6. Problem solving: draw a picture
7. Structures
8. Pointers and structures

C and C++ Strings

C++ has two mechanisms for manipulating strings.

The **`string`** class

- Supports character sequences of arbitrary length.
- Provides convenient operations: concatenation (+), comparison (`==, <, >`)

C strings

- Are arrays of type `char`
- Provide a more primitive level of string handling.
- Are from the C language (C++ was built from C).

`char` Type

The type **`char`** is an individual character.

```
char yes = 'y';
char no = 'n';
char maybe = '?';

char three = '3';    // not binary 3,
// but the printer-printable ASCII char for
// the numeral.  It happens to be binary
// 0110 0011 = 51 decimal!
```

Special Characters

`'\n'`: **newline**

`'\a'`: *alert* character – rings the bell

`'\t'`: **horizontal tab**

`'\0'`: **null character (binary zero)**

Etc…

These are still single (individual) characters:

the ***escape sequence*** characters. They control functions for a display or printer.

Character Literal Examples: Table 3

`'y'`	The character y
`'0'`	The character for the digit 0. In the ASCII code, '0' has the value 48.
`' '`	The space character
`'\n'`	The newline character
`'\t'`	The tab character
`'\0'`	The null terminator of a string
`"y"`	**Error:** Not a char value, but a `char` array of 2 characters (includes a string terminator)

The Null Terminator Character and C Strings

The null character is special to C strings because it is always the last character in them:

`"CAT"` is really 4 characters, not 3:

```
'C' 'A' 'T' '\0'
```

The null terminator character indicates the end of a C string.

Literal strings are always stored as character arrays.

Character Arrays as Storage for C Strings

As with all arrays, a string literal can be assigned to a pointer variable that points to the initial character in the array:

```
char* char_pointer = "Harry";
      // Points to the 'H'
```

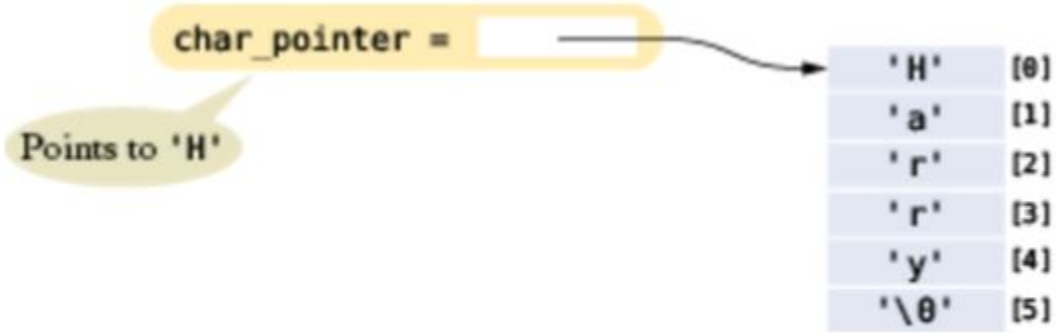

Figure 6 A Character Array

Using the Null Terminator Character

Functions that operate on C strings rely on this terminator.

The **`strlen`** function (from the `<cstring>` library) returns the length of a C string. Here is the source:

```
int strlen(const char s[])
{
   int i = 0;
   // Count characters before
   // the null terminator
   while (s[i] != '\0') { i++; }
   return i;
}
```

The call **`strlen("Harry")`** returns 5. The null terminator character is not counted as part of the "length" of the C string

Character Arrays

If you want to modify the characters in a C string, define a character array to hold the characters.

For example:

```
// An array of 6 characters
char c[] = "Harry";
```

You can modify the characters in the array:

```
c[0] = 'L'; //now c is Larry
```

Converting Between C and C++ Strings

The **cstdlib** header includes the function:

```
int atoi(const char s[])
```

The **atoi** function returns an `int` equivalent to a character array
containing digits:

```
char* year = "2012";
int y = atoi(year);
```

y is the integer **2012**

`c_str()` Function converts C++ `string` to a C string

Older versions of the C++ `<string>` library lack an `atoi()`.

The **`c_str`** member function offers an "escape hatch", converting the C++ `string` to a `char` array:

```
string year = "2012";
int y = atoi(year.c_str());
```

Again, **`y`** is the integer **`2012`**

Converting a C string to a C++ `string`

Converting from a C string to a C++ string is easy – the assignment operator (=) does it:

```
string name = "Harry";
Char* fred = "Fredrick";
string name2 = fred;
```

C++ Strings and the [] Operator

You can access individual characters in either a C-string or C++ string with the `[]` operator:

```
string name = "Harry";
name[3] = 'd'; //name now is Hardy

char c[] = "Mary";
c[3] = 'k'; // now is Mark
```

Example: Converting Case of a C++ `string` with `toupper()`

The **`toupper`** function is defined in the **`<cctype>`** header. It converts a lowercase `char` to uppercase. The **`tolower`** function does the opposite.

You can write a function that will return the uppercase version of a C++ **`string`**:

```
/**
   Makes an uppercase version of a string.
   @param str a string
   @return a string with the characters in str converted to
   uppercase
*/
string uppercase(string str)
{
   string result = str; // Make a copy of str
   for (int i = 0; i < result.length(); i++)
   {
      // Convert each character to uppercase
      result[i] = toupper(result[i]);
   }
   return result;
}
```

C String Functions from the <cstring> Library: Table 4

In this table, s and t are character arrays; n is an integer.	
Function	Description
`strlen(s)`	Returns the length of s.
`strcpy(t, s)`	Copies the characters from s into t.
`strncpy(t, s, n)`	Copies at most n characters from s into t.
`strcat(t, s)`	Appends the characters from s after the end of the characters in t.
`strncat(t, s, n)`	Appends at most n characters from s after the end of the characters in t.
`strcmp(s, t)`	Returns 0 if s and t have the same contents, a negative integer if s comes before t in lexicographic order, a positive integer otherwise.

C++ strings are usually easier than the <cstring> Functions

Consider the task of concatenating 2 names into a string. The string class makes this easy:

```
string first = "Harry";
string last = "Smith";
string name = first + " " + last;
```

With C strings, it is much harder, as we have to worry about sizes of the arrays:

```
const int NAME_SIZE = 40;
char name[NAME_SIZE];
strncpy(name, first, NAME_SIZE - 1);
int length = strlen(name);
if (length < NAME_SIZE - 1)
{
   strcat(name, " ");
   int n = NAME_SIZE - 2 - length;
     // Leave room for space, null terminator
   if (n > 0)
   {
      strncat(name, last, n);
   }
}
```