# Null terminated char arrays and cstring

- we very often want to read/write text in programs
- arrays of characters are a natural way to store text
- arrays need to be declared of some fixed size, but when we ask user to type in a name/word/sentence/etc we don't know how much text they will enter
- we create an array we think/hope  is big enough, fill the beginning of it with text they enter, then add a special character as a marker, showing the end of what they typed
- anything in the array after the marker isn't “in use” at the moment

# ONLY FOR ARRAYS OF CHAR

- the things we are discussing today are based on a widely-accepted convention for working with arrays of char
  - they do not work on other kinds of arrays (int, float, etc)
  - *they do not work on other kinds of arrays (int, float, etc)*
  - **they do not work on other kinds of arrays (int, float, etc)**
  - they do not work on other kinds of arrays (int, float, etc)
  - they do not work on other kinds of arrays (int, float, etc)
  - ...

# The null terminator

- characters in C++ are represented using the ascii character codes (see www.asciitable.com for list)
- 128 possible characters, each with its own corresponding integer code (from 0 to 127)
- most characters we can type at the keyboard are in range 32-127
- the character chosen as the special marker in char arrays is the one with ascii code 0, aka NULL
- can specify any character through it's ascii code using '\xxx', e.g. '\0' for NULL, '\32' for space, '\65' for “A', etc

# Input with null terminators

- cin and scanf can read a word into a char array
  - skips whitespace in front then reading the “word” as all characters before the next whitespace
  - adds a '\0' as the first character after what was read/stored

```
char arrayname[SIZE];
scanf(“%s”, arrayname);  // note no & used
cin >> arrayname;
```

- risk: neither one checks to make sure the text + null will actually fit in the given array

# Reading entire lines

- sometimes want to read a whole line, including whitespace
- <cstdio> we use fgets, specifying stdin as input source

```
char text[SIZE];
fgets(text, size, stdin);
```

- <iostream> we use getline, specifying cin as input source

```
cin.getline(text)
```

- or, with slightly different syntax, read into a C++ string

```
string s;
getline(cin, s);
```

# Not skipping whitespace

- sometimes we want to read a character without skipping whitespace
- <cstdio> we can use the getc function

```
char ch = getc(stdin);
```

- <iostream> uses the noskipws flag

```
cin >> noskipws >> ch;
```

# Output with null terminators

- cout and printf can each display contents of character array, assuming that there is a '\0' present indicating where to stop the output

```
cout << arrayname;
printf("%s", arrayname);
```

- risk: if no '\0' is present then they go "off the end" of the array, and keep printing until they happen to hit a byte in memory containing a 0

# Manually writing null term strings

- we can explicitly write content for a null terminated string:

```
arr[0] = 'a';
arr[1] = 'b';
arr[2] = ' ';
arr[3] = 'x';
arr[4] = '\0';
cout << arr;  // prints “ab x”
arr[0] = '\0';
cout << arr; // prints nothing, an empty string
```

# chars and ++, --

- ++, -- work on chars too, e.g.
- char c = 'a';
- c++; // c now has 'b'
- uses the ascii codes to decide which char is next (e.g. 'a' has code 65, 'b' has code 66

```
char text[15];
int i = 0;
char c = 'a';

while (i < 10) {
   text[i] = ch;
   i++;        // moves to next arr pos
   ch++;      // switch to next ascii char
}

// note i is 10 when we get out of loop
text[i] = '\0'; // puts null term in next spot
cout << text << endl;
// displays abcdefghij
```

# cstring library

- the <cstring> library provides a variety of functions that work on null-terminated arrays of char
  - assuming str, str1, str2 are arrays of char, and N is array size
  - strlen(str) // returns count of # chars before the '\0'
  - strcpy(str1,str2) // copies str2 into str1
  - strcat(str1,str2) // copies str2 onto end of str1
  - strncpy(str1,str2,N) // strcpy but at most N chars
  - strncat(str1, str2, N) // strcat but at most N chars
- each of them correctly adds the '\0' in right spot

# Example:

```
const int SIZE = 64;
char name1[SIZE];
char name2[SIZE];
char fullname[SIZE];

cout << “Enter first name”;
cin >> name1;                          // suppose they enter “scoobert”
cout << “Enter second name”;
cin >> name2;                           // suppose they enter “doo”

strcpy(fullname, name1);    // fullname is now “scoobert” then a '\0'
strcat(fullname, “, ”);           // fullname now “scoobert, “ then a '\0'
strcat(fullname, name2);     // fullname now “scoobert, doo” then a '\0'
cout << “The full name is: “ << fullname; // prints “scoobert, doo”
int L = strlen(fullname);  // L would be 13, counts all the characters before the null
```

# strcmp, strncmp

- we can also compare text “alphabetically” (actually using the order characters appear in the ascii table)
- strcmp(str1, str2) returns 0 if the contents are the same (up to the null terminator), a negative number if str1 comes before str2 “alphabetically”, a positive number otherwise
- strncmp(str1, str2, n) similar, but only checks first n chars

```
const int size = 10;
char str1[size];
char str2[size];

cin >> str1 >> str2;

if (strcmp(str1, str2) == 0) {
   cout << “they are the same” << endl;
} else if (strcmp(str1, str2) < 0) {
   cout << str1 << “ comes first” << endl;
} else {
   cout << str2 << “ comes first” << endl;
}
```