Strings and Pointers in C

String Declarations

- Unlike Java, there is no String data type in C
- Strings in C are simply arrays of characters terminated with 0 (character ‘\0’)

```c
char some[10]; // need to specify max size

// one way:
char msg[6] = {'H','e','y',' ','!','\0'};

// another way (last 2 places unused):
char msg[8] = "Hey !"; // no '\0'

// or
char msg[] = "Hey !"; // no '\0'
```

memory for 6 characters (5 plus the null char ‘\0’) automatically allocated
**Memory Representation**

- `char some[10];`

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
```

- `char msg[6] = {'H', 'e', 'y', ' ', '!', '\0'};`

```
[0] [1] [2] [3] [4] [5]  
'H' 'e' 'y' ' ' '!' '\0'  
```

- `char msg[8] = "Hey !";`

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]  
'H' 'e' 'y' '!' '!' '!' '\0'  
```

- `char msg[] = "Hey !";`

```
[0] [1] [2] [3] [4] [5]  
'H' 'e' 'y' '!' '!' '!' '\0'  
```

---

**Reading Into a String (1)**

```c
#define MAX_BUFFER 20
char buffer[MAX_BUFFER];

scanf("%s", buffer); // or
gets(buffer, MAX_BUFFER);
```

- What if the array is not large enough to hold the input?
  - characters will be stored into memory locations past the end of the array
  - will result in run-time memory access violation error!
Reading Into a String (2)  

- Better:
  ```c
  #define MAX_BUFFER 20
  char buffer[MAX_BUFFER];

  fgets(buffer, MAX_BUFFER, stdin);
  ```

- `fgets` is similar to `gets`, but:
  - it takes a third argument, in our case standard input
  - it stores into buffer no more than MAX_BUFFER chars (extra characters are ignored), so memory violation error won’t occur

Functions for Manipulating Strings  

- C provides a large number of functions for manipulating strings. Four important ones:
  ```c
  strlen(s)
  // returns the length of s

  strcpy(toS, fromS)
  // copy fromS to toS (toS must be large enough)

  strcmp(s1, s2)
  // returns 0 if s1 == s2
  // returns an integer < 0 if s1 < s2
  // returns an integer > 0 if s1 > s2

  strtok – read the Sun manual pages to find out what this function does
  ```
What are Pointers?

- A pointer is a variable that holds the address of another variable (object).

- Suppose that we have an integer variable
  
  ```c
  int i;
  ```

  and wish to have a pointer point to this variable. Thus we need to know the memory address of `i`.

- How do we know the address of `i`? ...

  ```c
  &i
  ```

  is the address of `i`. The operator `&` is called the address-of operator.
We can declare that a pointer `iPtr` points to an `int` by saying

```
int * iPtr;
```

Suppose that we have:

```
int i = 5;
int j = 7;
```

We can make `iPtr` point to `i` by assigning to `iPtr` the memory location where `i` is stored. Thus

```
iPtr = &i;
```

sets `iPtr` to point to `i`.

---

We can also initialize `iPtr` at the point of declaration:

```
int i;
int * iPtr = &i;
```

Here is a common error:

```
int i;
int * iPtr = i;  // ERROR: i is not an address
```
Declaring Pointers

- When declaring several pointer variables in one statement - the asterisk does not distribute throughout the statement:

  \[
  \begin{align*}
  \text{int } & \ast p, q; & \quad \text{equivalent to} & \quad \text{int } \ast p; \\
  \text{int } & \ast p, \ast q; & \quad \text{equivalent to} & \quad \text{int } \ast p; \quad \text{int } \ast q;
  \end{align*}
  \]

Dereference *

- The value of the data being pointed at is obtained by using the operator \( \ast \)

- If \( p \) is a pointer value, then

  \[
  \ast p
  \]

  refers to the variable pointed to by \( p \). Since reference is another name for address, the operator \( \ast \) is called dereference operator.
**Dereference Example**

\[
\begin{align*}
\text{int } & \ i; \quad \text{int } * \ p = &i; \quad \star p = 101; \quad \star p++;
\end{align*}
\]

```
int i;
int * p = &i;
*p = 101;
(*p)++;
```

- A dereferenced pointer behaves exactly like the variable it points to.

**Note the Difference ...**

```
ptr1 i 5
ptr2 j 7
```

```
ptr1 i 5
ptr2 j 7
```

```
ptr1 i 7
ptr2 j 7
```

- Equivalent to \( i = 101; \)
- Equivalent to \( i++; \)

- Initial state
- After \( \text{ptr1} = \text{ptr2}; \) starting from initial state
- After \( \*\text{ptr1} = \*\text{ptr2}; \) starting from initial state
Uninitialized Pointers

Suppose that we have the following declarations:

```c
int i;
int * iPtr;
*iPtr = 100;
```

What is the value of `iPtr`? **Undefined.** What could happen?

- `iPtr` could hold an address that does not make sense at all, causing your program to crash if dereferenced.
- `iPtr` could point to an address which is accessible. Then the assignment `*iPtr = 100;` would accidentally change some other data, which could result in a crash at a later point. This is a tough error to detect since the cause and symptom may be widely separated in time.

---

Putting it all Together...

... with the help of a simple example:

```c
int i, value;
int * iPtr; // declares iPtr to be a pointer to an integer
i = 510; // Step 1
iPtr = &i; // Step 2
value = *iPtr; // Step 3
```

<table>
<thead>
<tr>
<th>Step</th>
<th>iPtr</th>
<th>i</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1</strong></td>
<td>???</td>
<td>510</td>
<td>???</td>
</tr>
<tr>
<td><strong>Step 2</strong></td>
<td>iPtr</td>
<td>510</td>
<td>???</td>
</tr>
<tr>
<td><strong>Step 3</strong></td>
<td>iPtr</td>
<td>510</td>
<td>510</td>
</tr>
</tbody>
</table>
The null Pointer

- The value of a pointer can be:
  - some garbage (pointer unassigned)
  - the address of some variable (e.g., `int * p = &i;`)
  - the constant 0 (the null pointer, points to absolutely nothing)

```c
somePointer = 0;
```

This statement does not cause `somePointer` to point to memory location zero; it guarantees that `somePointer` does not point to anything.

- The null pointer is a special pointer value that a program can test for:

```c
if (somePointer == 0) ...
```

Strings and Pointers

- A string name is basically a constant pointer

```c
char a[3];
```

Consider the declaration:

```
&a[0] 1000
&a[1] 1001
&a[2] 1002
...
&a[2] 2160
```

The compiler allocates three characters for the array object. These are referenced as `a[0]`, `a[1]`, `a[2]` and occupy a contiguous block of memory.

- The value of `a` is exactly `&a[0]`, the address of the first integer in the array
Arrays and Pointers - Examples

Consider the following declarations:

```c
char a[5] = "123";
int * p;
p = &a[2];
```

You can use the index [] operator with a pointer:

```c
p[0] = '4';
p[1] = '5';
```

Indexing can be used with any pointer, but it only makes sense when the pointer points to an array.

Arrays are NOT Pointers

Declaring an array sets aside space for its elements

```c
char a[5];
```

Declaring a pointer variable sets aside only space to hold the variable

```c
char * p;
```

You can change a pointer variable, but not the address of an array

```c
char b[6];
p = b;    // OK
a = b;    // ERROR!
```
Other resources

- A very good tutorial on pointers and arrays in C:

  http://pw1.netcom.com/%7Etjensen/ptr/pointers.htm