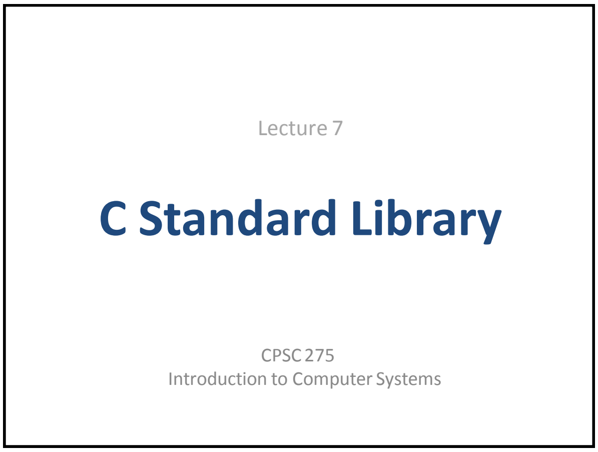
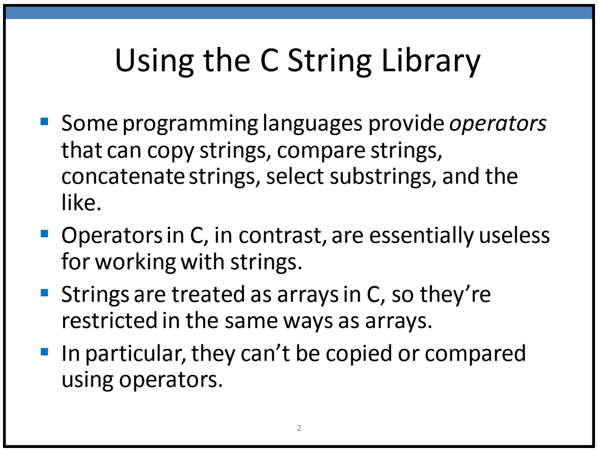


Lecture 7

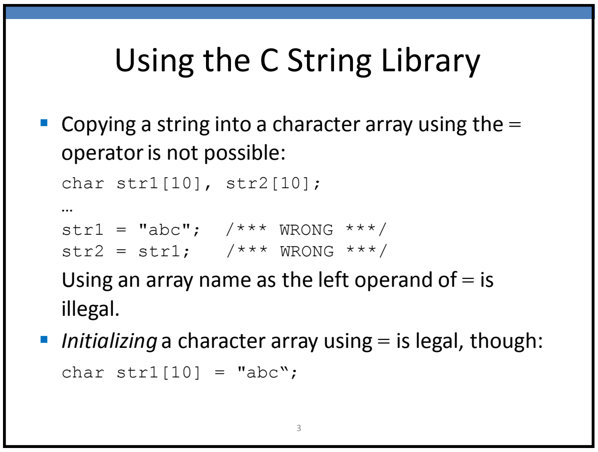
C Standard Library

CPSC 275
Introduction to Computer Systems



Using the C String Library

- Some programming languages provide *operators* that can copy strings, compare strings, concatenate strings, select substrings, and the like.
- Operators in C, in contrast, are essentially useless for working with strings.
- Strings are treated as arrays in C, so they're restricted in the same ways as arrays.
- In particular, they can't be copied or compared using operators.



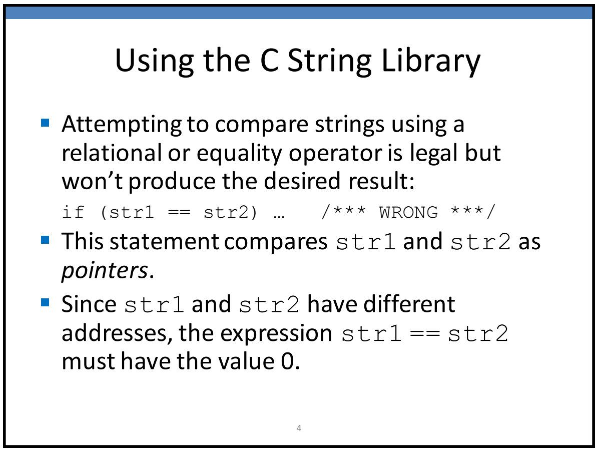
Using the C String Library

- Copying a string into a character array using the = operator is not possible:

  ```
  char str1[10], str2[10];
  …
  str1 = "abc";   /*** WRONG ***/
  str2 = str1;    /*** WRONG ***/
  ```

  Using an array name as the left operand of = is illegal.
- *Initializing* a character array using = is legal, though:

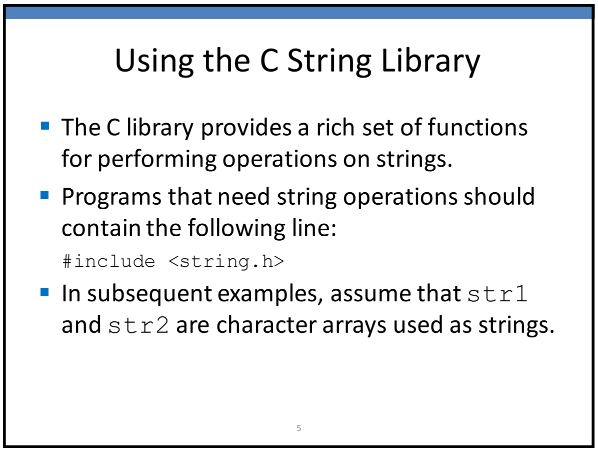
  ```
  char str1[10] = "abc“;
  ```



Using the C String Library

- Attempting to compare strings using a relational or equality operator is legal but won't produce the desired result:

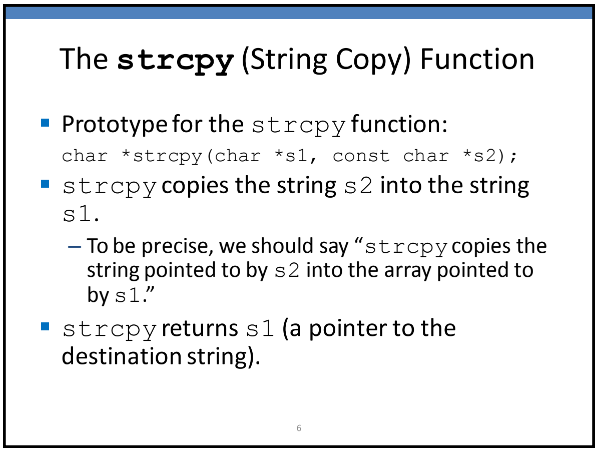
  ```
  if (str1 == str2) …   /*** WRONG ***/
  ```
- This statement compares `str1` and `str2` as *pointers*.
- Since `str1` and `str2` have different addresses, the expression `str1 == str2` must have the value 0.



Using the C String Library

- The C library provides a rich set of functions for performing operations on strings.
- Programs that need string operations should contain the following line:

  ```
  #include <string.h>
  ```
- In subsequent examples, assume that `str1` and `str2` are character arrays used as strings.



The **`strcpy`** (String Copy) Function

- Prototype for the `strcpy` function:

  ```
  char *strcpy(char *s1, const char *s2);
  ```
- `strcpy` copies the string `s2` into the string `s1`.
  - To be precise, we should say "`strcpy` copies the string pointed to by `s2` into the array pointed to by `s1`."
- `strcpy` returns `s1` (a pointer to the destination string).

The **strcpy** (String Copy) Function

- A call of `strcpy` that stores the string `"abcd"` in `str2`:

```
strcpy(str2, "abcd");
  /* str2 now contains "abcd" */
```

- A call that copies the contents of `str2` into `str1`:

```
strcpy(str1, str2);
  /* str1 now contains "abcd" */
```

The **strcpy** (String Copy) Function

- In the call `strcpy(str1, str2)`, `strcpy` has no way to check that the `str2` string will fit in the array pointed to by `str1`.
- If it doesn't, undefined behavior occurs.

The **strcpy** (String Copy) Function

- Calling the `strncpy` function is safer.
- `strncpy` has a third argument that limits the number of characters that will be copied.
- A call of `strncpy` that copies `str2` into `str1`:

```
strncpy(str1, str2, sizeof(str1));
```

The **strcpy** (String Copy) Function

- What if the length of `str2` is greater than or equal to the size of the `str1` array?
- A safer way to use `strncpy`:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

- The second statement guarantees that `str1` is always null-terminated.

The **strlen** (String Length) Function

- Prototype for the `strlen` function:

```
size_t strlen(const char *s);
```

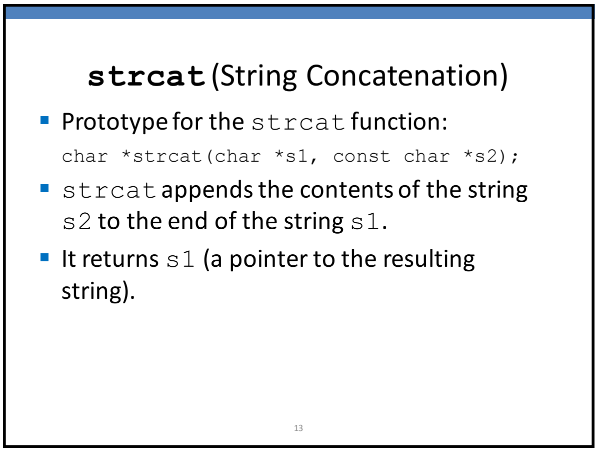
- `size_t` is a `typedef` name that represents one of C's unsigned integer types.

The **strlen** (String Length) Function

- `strlen` returns the length of a string `s`, not including the null character.
- Examples:

```
int len;

len = strlen("abc");   /* len is now 3 */
len = strlen("");      /* len is now 0 */
strcpy(str1, "abc");
len = strlen(str1);    /* len is now 3 */
```

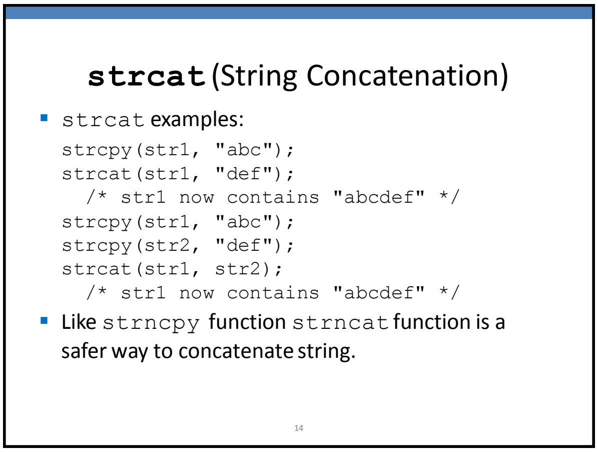


`strcat` (String Concatenation)

- Prototype for the `strcat` function:

```
char *strcat(char *s1, const char *s2);
```

- `strcat` appends the contents of the string `s2` to the end of the string `s1`.
- It returns `s1` (a pointer to the resulting string).

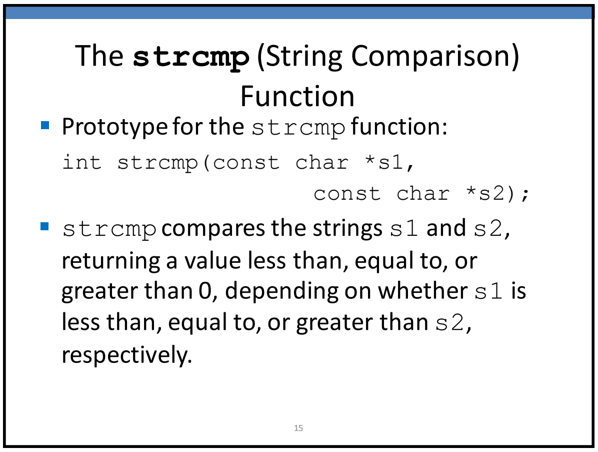


`strcat` (String Concatenation)

- `strcat` examples:

```
strcpy(str1, "abc");
strcat(str1, "def");
  /* str1 now contains "abcdef" */
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);
  /* str1 now contains "abcdef" */
```

- Like `strncpy` function `strncat` function is a safer way to concatenate string.

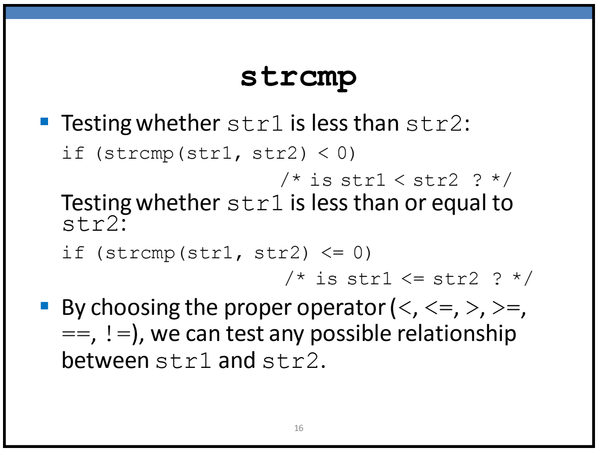


The **`strcmp`** (String Comparison) Function

- Prototype for the `strcmp` function:

```
int strcmp(const char *s1,
                     const char *s2);
```

- `strcmp` compares the strings `s1` and `s2`, returning a value less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2`, respectively.



`strcmp`

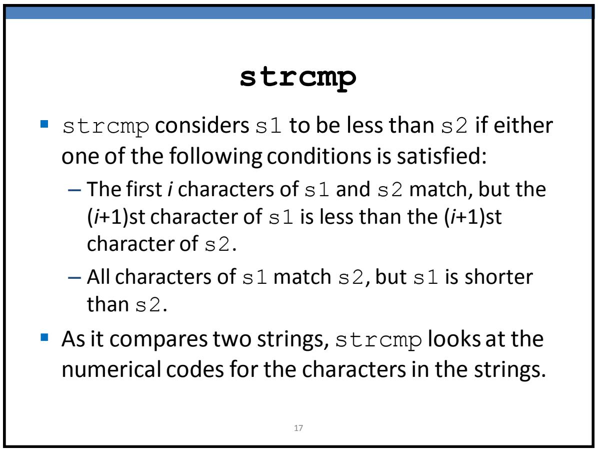
- Testing whether `str1` is less than `str2`:

```
if (strcmp(str1, str2) < 0)
                         /* is str1 < str2 ? */
```

Testing whether `str1` is less than or equal to `str2`:

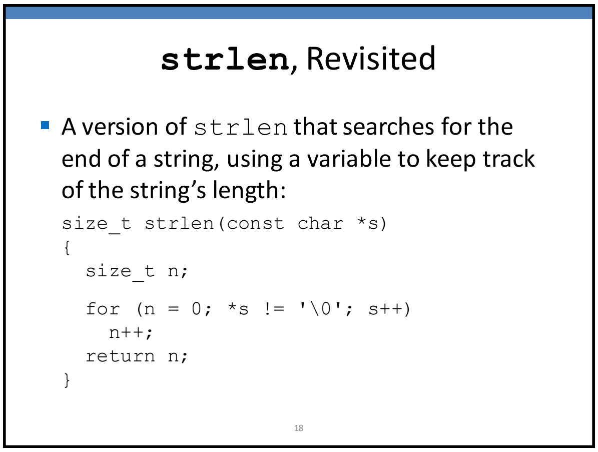
```
if (strcmp(str1, str2) <= 0)
                         /* is str1 <= str2 ? */
```

- By choosing the proper operator (<, <=, >, >=, ==, !=), we can test any possible relationship between `str1` and `str2`.



`strcmp`

- `strcmp` considers `s1` to be less than `s2` if either one of the following conditions is satisfied:
  - The first *i* characters of `s1` and `s2` match, but the (*i*+1)st character of `s1` is less than the (*i*+1)st character of `s2`.
  - All characters of `s1` match `s2`, but `s1` is shorter than `s2`.
- As it compares two strings, `strcmp` looks at the numerical codes for the characters in the strings.

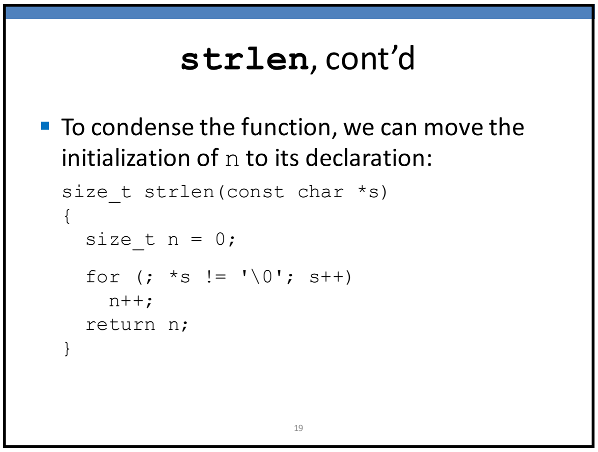


`strlen`, Revisited

- A version of `strlen` that searches for the end of a string, using a variable to keep track of the string's length:

```
size_t strlen(const char *s)
{
  size_t n;

  for (n = 0; *s != '\0'; s++)
    n++;
  return n;
}
```

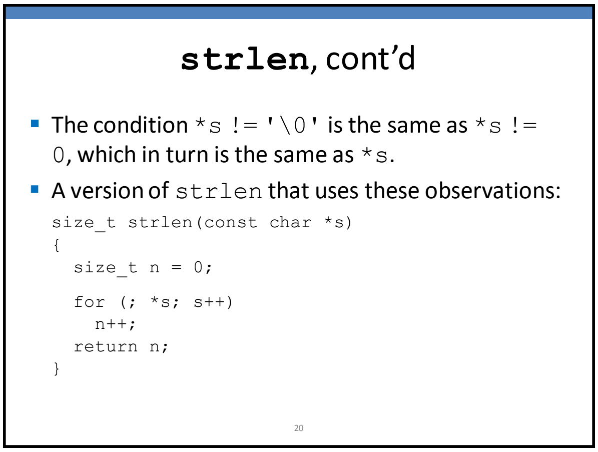


strlen, cont'd

- To condense the function, we can move the initialization of `n` to its declaration:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s != '\0'; s++)
    n++;
  return n;
}
```

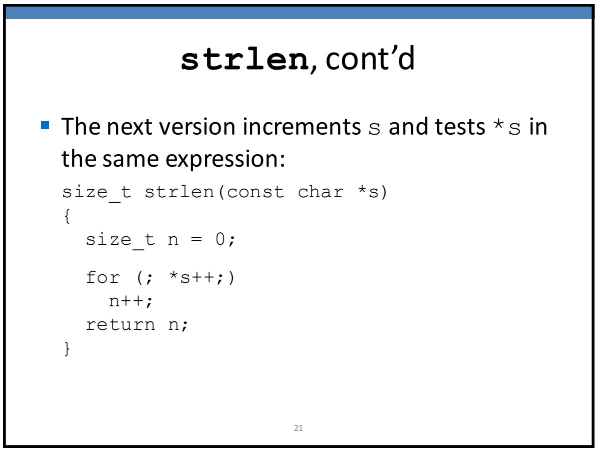


strlen, cont'd

- The condition `*s != '\0'` is the same as `*s != 0`, which in turn is the same as `*s`.
- A version of `strlen` that uses these observations:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s; s++)
    n++;
  return n;
}
```

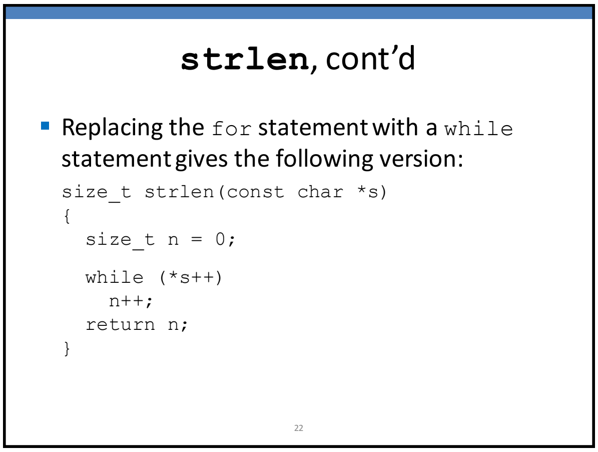


strlen, cont'd

- The next version increments `s` and tests `*s` in the same expression:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s++;)
    n++;
  return n;
}
```

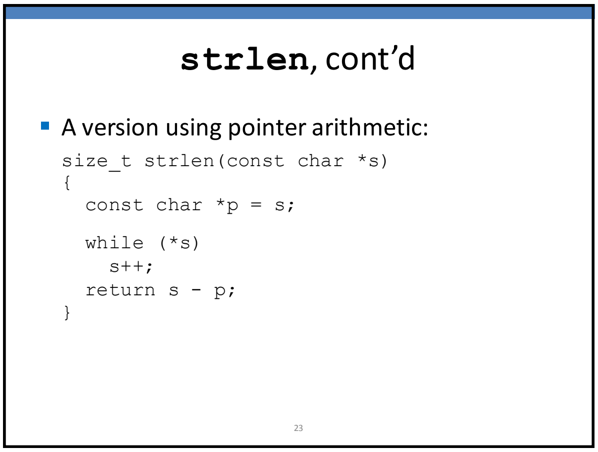


strlen, cont'd

- Replacing the `for` statement with a `while` statement gives the following version:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  while (*s++)
    n++;
  return n;
}
```

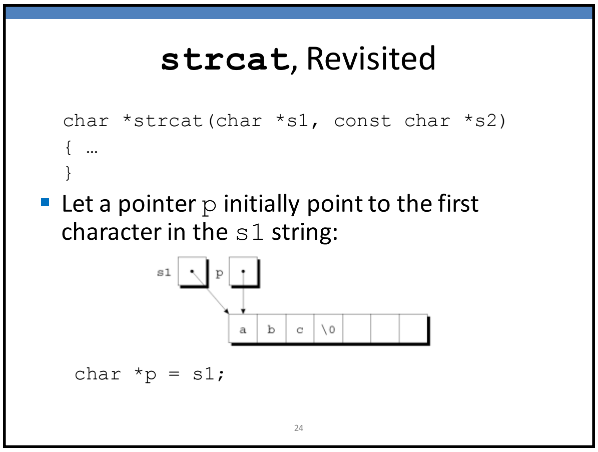


strlen, cont'd

- A version using pointer arithmetic:

```
size_t strlen(const char *s)
{
  const char *p = s;

  while (*s)
    s++;
  return s - p;
}
```



strcat, Revisited

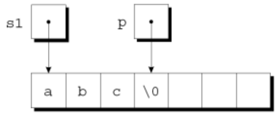
```
char *strcat(char *s1, const char *s2)
{ …
}
```

- Let a pointer `p` initially point to the first character in the `s1` string:

```
char *p = s1;
```

strcat, cont'd

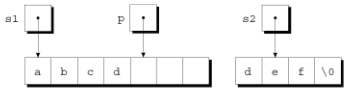
- Locate the null character at the end of the string s1 and make p point to it.



```
while (*p != '\0')
   p++;
```

strcat, cont'd

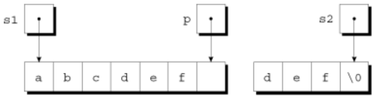
- Copy characters of s2 one at a time.
- The strings after copying the first character:



```
while (*s2 != '\0') {
   *p = *s2;
   p++;
   s2++;
}
```

strcat, cont'd

- After copying all of characters in s2:



- Are we done?

strcat, cont'd

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;
  while (*p != '\0')
    p++;
  while (*s2 != '\0') {
    *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';  /* terminate with a null */
  return s1;
}
```

A Condensed Version of **strcat**

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p)
    p++;
  while (*p++ = *s2++)
    ;  /* do nothing */
  return s1;
}
```

Arrays of Strings

- There is more than one way to store an array of strings.
- One option is to use a two-dimensional array of characters, with one string per row:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

- The number of rows in the array can be omitted, but we must specify the number of columns.

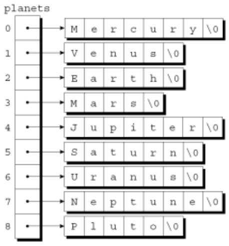
Arrays of Strings

- Unfortunately, the `planets` array contains a fair bit of wasted space (extra null characters):

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

Arrays of Strings Using Pointers

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```



Arrays of Strings

- To access one of the planet names, all we need do is subscript the `planets` array.
- Accessing a character in a planet name is done in the same way as accessing an element of a two-dimensional array.
- A loop that searches the `planets` array for strings beginning with the letter M:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```