String Idioms

- Functions that manipulate strings are a rich source of idioms and tricks
- We will explore some of the most famous idioms by using them to write
  - the `strlen` function and
  - the `strcat` function

Searching for the End of a String

- 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;
}
```

Searching for the End of a String

- 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;
}
```


Searching for the End of a String

- 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;
}
```

Searching for the End of a String

- 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;
}
```


Searching for the End of a String

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

```
size_t strlen(const char *s)
{
  size_t n = 0;
  while (*s++)
    n++;
  return n;
}
```

Searching for the End of a String

- Although we have condensed `strlen` quite a bit, it is likely that we have not increased its speed
- A version that *does* run faster, at least with some compilers:

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

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


Searching for the End of a String

- Idioms for "*search for the null character at the end of a string*":

```
while (*s)       while (*s++)
  s++;             ;
```

- The first version leaves `s` pointing to the null character
- The second version is more concise, but leaves `s` pointing just past the null character

Searching for the End of a String

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

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

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

   while (*s++)
     ;
   return s - p - 1;
}
```

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

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

 return n;
}
```


Concatenating Strings

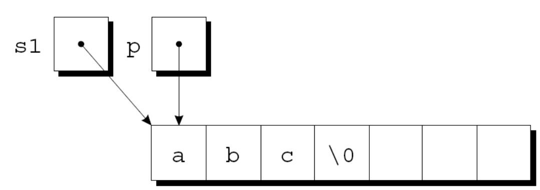
- Concatenating two strings is another common operation
- To introduce C's "string concatenate" idiom, we'll develop two versions of the `strcat` function
- The first version of `strcat` (next slide) uses a two-step algorithm:
  - Locate the null character at the end of the string `s1` and make `p` point to it
  - Copy characters one by one from `s2` to where `p` is pointing

Concatenating Strings

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

  while (*p != '\0')
    p++;
  while (*s2 != '\0')
  { *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';
  return s1;
}
```

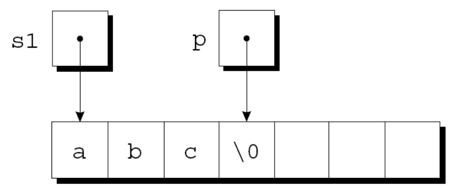

Concatenating Strings

- `p` initially points to the first character in the `s1` string:



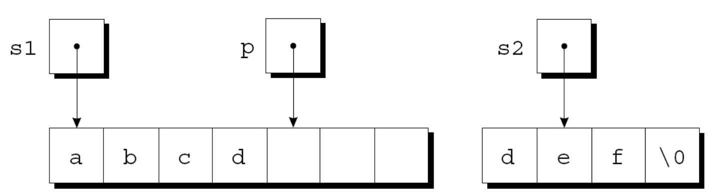
Concatenating Strings

- The first `while` statement locates the null character at the end of `s1` and makes `p` point to it:



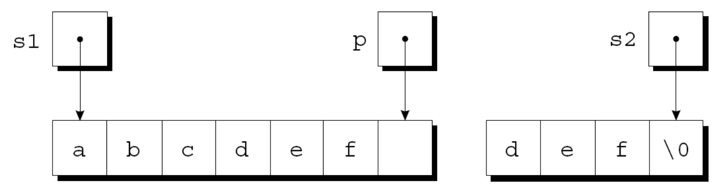
Concatenating Strings

- The second `while` statement repeatedly copies one character from where `s2` points to where `p` points, then increments both `p` and `s2`.
- Assume that `s2` originally points to the string `"def"`.
- The strings after the first loop iteration:



Concatenating Strings

- The loop terminates when `s2` points to the null character:



- After putting a null character where `p` is pointing, `strcat` returns

Concatenating Strings

- Condensed version of `strcat`:

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

  while (*p)
    p++;
  while (*p++ = *s2++)
    ;
  return s1;
}
```

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

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

*p = '\0';
```

Concatenating Strings

- The heart of the streamlined `strcat` function is the "string copy" idiom:
```
while (*p++ = *s2++)
  ;
```
- Ignoring for a second the two `++` operators, the expression inside the parentheses is an assignment:
```
*p = *s2
```
- After the assignment, `p` and `s2` are incremented
- Repeatedly evaluating this expression copies characters from where `s2` points to where `p` points
- But what causes the loop to terminate?

Concatenating Strings

- The `while` statement tests the character that was copied by the assignment `*p = *s2`
- All characters except the null character test `true`
- The loop terminates ***after*** the assignment, so the `null` character will be copied

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

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

- 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

The data in this 2D array are stored in a contiguous memory location (row after row), i.e., planets[2][0] is located immediately after planets[1][7].

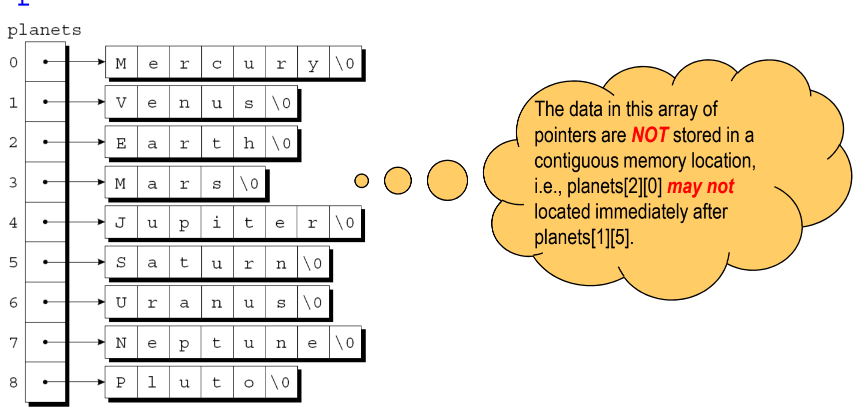
Arrays of Strings

- Most collections of strings will have a mixture of long strings and short strings
- What we need is a ***ragged array,*** whose rows can have different lengths
- We can simulate a ragged array in C by creating an array whose elements are *pointers* to strings:

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


Arrays of Strings

- This small change has a dramatic effect on how `planets` is stored:



Arrays of Strings

- To access one of the planet names, all we need to 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]);
```


Command-Line Arguments

- When we run a program, we'll often need to supply it with information
- This may include a file name or a switch that modifies the program's behavior
- Examples of the Unix `ls` command:

```
ls
ls -l
ls -l remind.c
```

Command-Line Arguments

- Command-line information is available to all programs, not just operating system commands
- To obtain access to ***command-line arguments***, `main` must have two parameters:

```
int main(int argc, char *argv[])
{
  …
}
```

- Command-line arguments are called ***program parameters*** in the **C** standard

Command-Line Arguments

- `argc` ("*argument count*") is the number of command-line arguments, *including name of the program*
- `argv` ("*argument vector*") is an array of pointers to the command-line arguments (**stored as strings**)

- The number of elements in the `argv` array is `argc + 1`
  - `argv[0]` points to the name of the program
  - `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments
  - `argv[argc]` is always a ***null pointer***

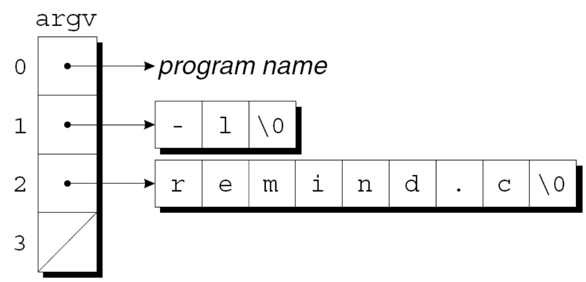
Command-Line Arguments

- If the user enters the command line

  `prog -l remind.c`

  then `argc` will be **3**, *not 2*, and

  `argv` array will have **4** elements, *not 3*, as follow:



Command-Line Arguments

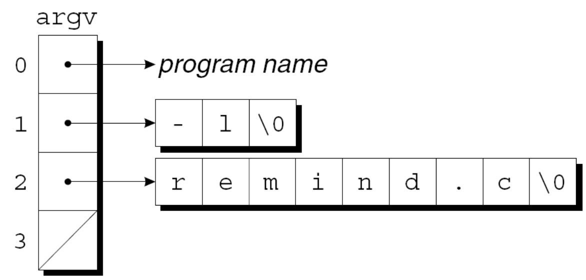
- Since `argv` is an array of pointers, accessing command-line arguments is easy
- Typically, a program that expects command-line arguments will set up a loop that examines each argument in turn
- One way to write such a loop is to use an integer variable as an index into the `argv` array:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

Command-Line Arguments

- Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly

```
char **p;

for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

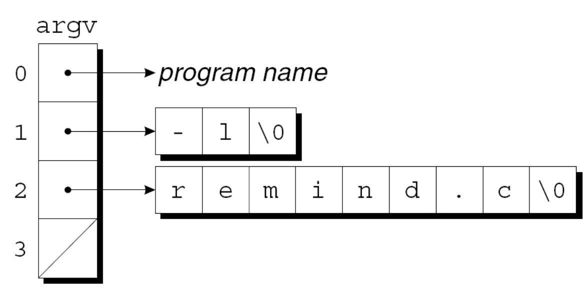


Chapter 13: Strings

Command-Line Arguments

- Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly

```
char **p;

for (p = &argv[1]; *p; p++)
  printf("%s\n", *p);
```



Program: Checking Planet Names

- The `planet.c` program illustrates how to access command-line arguments
- The program is designed to check a series of strings to see which ones are names of planets
- The strings are put on the command line:

```
planet Jupiter venus Earth fred
```

- The program will indicate whether each string is a planet name and, if it is, display the planet's number:

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```


planet.c

```
/* Checks planet names */
#include <stdio.h>
#include <string.h>

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{ int i, j;
  char *planets[] = {"Mercury", "Venus", "Earth", "Mars", "Jupiter",
                     "Saturn","Uranus", "Neptune", "Pluto"};

  for (i = 1; i < argc; i++)
  { for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0)
      { printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```

planet.c (Revised)

```c
/* Checks planet names */
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{ int i, j;
  char *planets[] = {"Mercury", "Venus", "Earth", "Mars", "Jupiter",
                     "Saturn","Uranus", "Neptune", "Pluto", ""};
  for (i = 1; i < argc; i++)
  { j=0;
    while (planets[j][0])
      if (strcmp(argv[i], planets[j++]) == 0)
      { printf("%s is planet %d\n", argv[i], j);
        break;
      }
    if (!planets[j][0])
      printf("%s is not a planet\n", argv[i]);
  }
  return 0;
}
```

What will happen if we entered "Pluto"

planet.c (Revised)

```c
/* Checks planet names */
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{ int i, j;
  char *planets[] = {"Mercury", "Venus", "Earth", "Mars", "Jupiter",
                     "Saturn","Uranus", "Neptune", "Pluto", ""};
  for (i = 1; i < argc; i++)
  { j=0;
    while (planets[j][0])
      if (strcmp(argv[i], planets[j++]) == 0)
      { printf("%s is planet %d\n", argv[i], j);
        j--;
        break;
      }
    if (!planets[j][0])
      printf("%s is not a planet\n", argv[i]);
  }
  return 0;
}
```

What will happen if we entered "Pluto"

planet.c (Revised 2)

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{ int i ;
  char **p, *planets[] = {"Mercury", "Venus", "Earth", "Mars", "Jupiter",
                          "Saturn","Uranus", "Neptune", "Pluto", ""};

  for (i = 1; i < argc; i++) {
    p = planets;
    while (**p)
      if (strcmp(argv[i], *p++) == 0)
      { printf("%s is planet %d\n", argv[i], p - planets);
        *p--;
        break;
      }
    if (!**p)
      printf("%s is not a planet\n", argv[i]);
  }
  return 0;
}
```