The C String Library

Using the C String Library

- Some programming languages provide operators that can copy strings, compare strings, concatenate strings, select substrings, and the like.
- C's operators, 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

- Direct attempts to copy or compare strings will fail.
- 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";
```

  In this context, = is not the assignment operator.

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 a safer, albeit slower, way to copy a string.
- `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

- `strncpy` will leave `str1` without a terminating null character 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 */
```

The **`strcat`** (String Concat.) Function

- 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` examples:

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

The **`strcat`** (String Concat.) Function

- As with `strcpy`, the value returned by `strcat` is normally discarded.
- The following example shows how the return value might be used:

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

The **`strcat`** (String Concat.) Function

- `strcat(str1, str2)` causes undefined behavior if the `str1` array isn't long enough to accommodate the characters from `str2`.
- Example:

```
char str1[6] = "abc";

strcat(str1, "def");    /*** WRONG ***/
```

- `str1` is limited to six characters, causing `strcat` to write past the end of the array.

The **`strcat`** (String Concat.) Function

- The `strncat` function is a safer but slower version of `strcat`.
- Like `strncpy`, it has a third argument that limits the number of characters it will copy.
- A call of `strncat`:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

- `strncat` will terminate `str1` with a null character, which isn't included in the third argument.

The **`strcat`** (String Concat.) 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`.

The `strcat` (String Concat.) Function

- 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`.

The **`strcat`** (String Concat.) Function

- `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`.

The `strcat` (String Concat.) Function

- As it compares two strings, `strcmp` looks at the numerical codes for the characters in the strings.
- Some knowledge of the underlying character set is helpful to predict what `strcmp` will do.
- Important properties of ASCII:
  - A–Z, a–z, and 0–9 have consecutive codes.
  - All upper-case letters are less than all lower-case letters.
  - Digits are less than letters.
  - Spaces are less than all printing characters.

Writing String Functions

- We'll explore some details of writing the `strlen` and `strcat` functions.

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've condensed `strlen` quite a bit, it's likely that we haven't 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.

Copying a String

- Copying a string is another common operation.
- To introduce C's "string copy" 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.

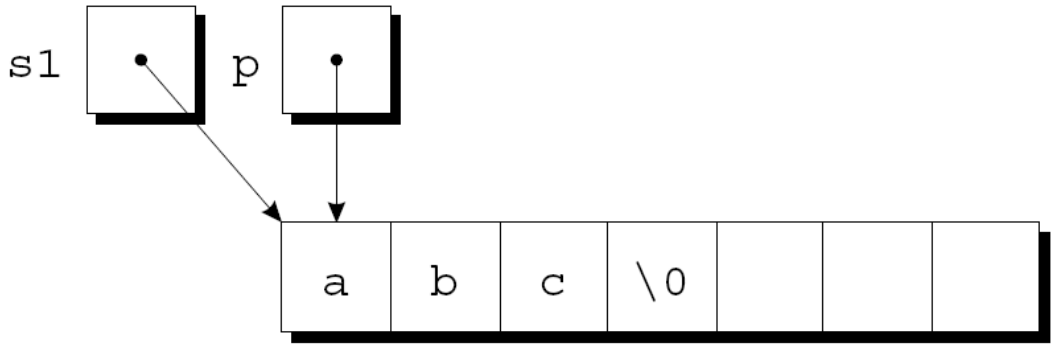
Copying a String

```
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;
}
```

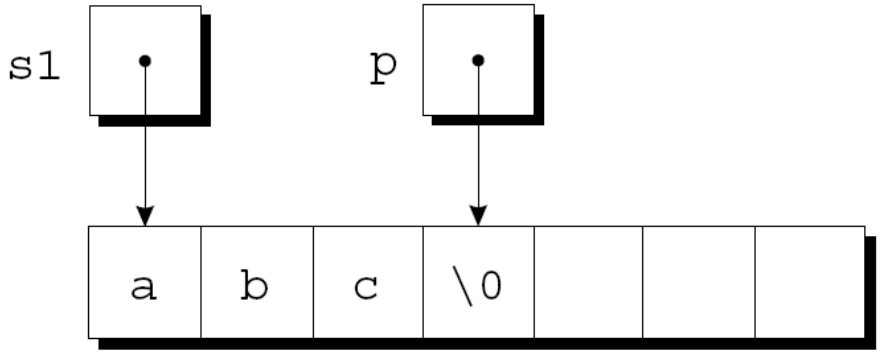
Copying a String

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



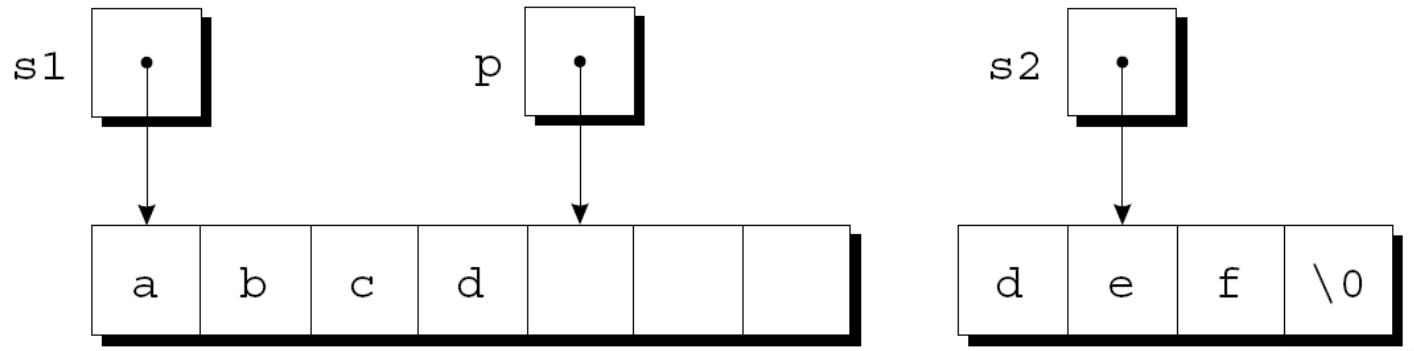
Copying a String

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



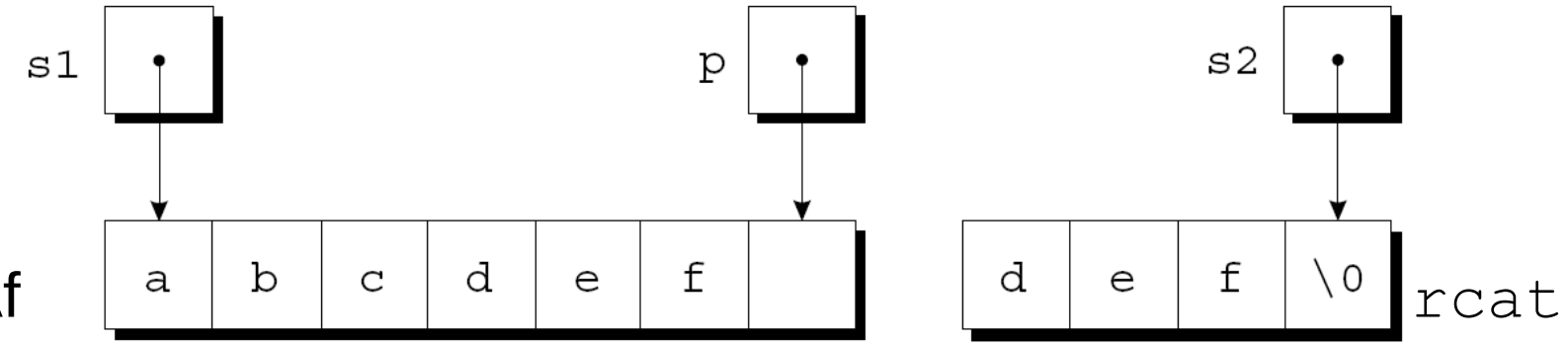
Copying a String

- 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:



Copying a String

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



- Af `rcat` returns.

Copying a String

- Condensed version of `strcat`:

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

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

Copying a String

- The heart of the streamlined `strcat` function is the "string copy" idiom:

```
while (*p++ = *s2++)
  ;
```

- Ignoring 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.

Copying a String

- But what causes the loop to terminate?
- 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.