ESc 101: Fundamentals of Computing

Lecture 24

Feb 25, 2010

Outline

Why is Size of Second Dimension Required in Parameter Declaration

Consider function `read_matrix()`:

```
read_vector(int vector[])
{
    for (int i = 0; i < SIZE; i++)
        scanf("%d", &vector[i]);
}

read_matrix(char *statement, int A[][SIZE])
{
    printf(statement);
    for (int i = 0; i < SIZE; i++)
        read_vector(A[i]);
}
```

Why is Size of Second Dimension Required in Parameter Declaration

- In `read_matrix()`, `read_vector(A[i])` is called several times.
- We can replace this by `read_vector(*(A+i))` as discussed earlier.
- `*(A+1)` is the same as `A[1]` which points to `A[1][0]`.
- That means `*(A+1)` shifts the pointer by `4*SIZE` bytes!
- For this reason, the second dimension must be provided in the parameter declaration.

Why is Size of Second Dimension Required in Parameter Declaration

- In `read_matrix()`, `read_vector(A[i])` is called several times.
- We can replace this by `read_vector(*(A+i))` as discussed earlier.
- `*(A+1)` is the same as `A[1]` which points to `A[1][0]`.
- That means `*(A+1)` shifts the pointer by `4*SIZE` bytes!
- For this reason, the second dimension must be provided in the parameter declaration.

Why is Size of Second Dimension Required in Parameter Declaration

- In `read_matrix()`, `read_vector(A[i])` is called several times.
- We can replace this by `read_vector(*(A+i))` as discussed earlier.
- `*(A+1)` is the same as `A[1]` which points to `A[1][0]`.
- That means `*(A+1)` shifts the pointer by `4*SIZE` bytes!
- For this reason, the second dimension must be provided in the parameter declaration.

Why is Size of Second Dimension Required in Parameter Declaration

- In `read_matrix()`, `read_vector(A[i])` is called several times.
- We can replace this by `read_vector(*(A+i))` as discussed earlier.
- `*(A+1)` is the same as `A[1]` which points to `A[1][0]`.
- That means `*(A+1)` shifts the pointer by `4*SIZE` bytes!
- For this reason, the second dimension must be provided in the parameter declaration.

Why is Size of Second Dimension Required in Parameter Declaration

- In `read_matrix()`, `read_vector(A[i])` is called several times.
- We can replace this by `read_vector(*(A+i))` as discussed earlier.
- `*(A+1)` is the same as `A[1]` which points to `A[1][0]`.
- That means `*(A+1)` shifts the pointer by `4*SIZE` bytes!
- For this reason, the second dimension must be provided in the parameter declaration.

Outline

String Operations

- Since strings are just arrays and are not treated in any special way, operations on strings are not provided in C.
- However, a library of functions for operating on strings exists and can be accessed by using the header declaration `#include <string.h>`.
- It provides the following operations:
  - `strcmp(s, t)`: compares two strings `s` and `t`.
  - `strcat(s, t)`: concatenates string `t` to `s`.
  - `strlen(s)`: computes the length of string `s`.
  - `strcpy(s, t)`: copies string `t` into `s`.

String Operations

- Since strings are just arrays and are not treated in any special way, operations on strings are not provided in C.
- However, a library of functions for operating on strings exists and can be accessed by using the header declaration `#include <string.h>`.
- It provides the following operations:
  - `strcmp(s, t)`: compares two strings `s` and `t`.
  - `strcat(s, t)`: concatenates string `t` to `s`.
  - `strlen(s)`: computes the length of string `s`.
  - `strcpy(s, t)`: copies string `t` into `s`.

String Operations

- Since strings are just arrays and are not treated in any special way, operations on strings are not provided in C.
- However, a library of functions for operating on strings exists and can be accessed by using the header declaration `#include <string.h>`.
- It provides the following operations:
  - `strcmp(s, t)`: compares two strings `s` and `t`.
  - `strcat(s, t)`: concatenates string `t` to `s`.
  - `strlen(s)`: computes the length of string `s`.
  - `strcpy(s, t)`: copies string `t` into `s`.

String Operations

- Since strings are just arrays and are not treated in any special way, operations on strings are not provided in C.
- However, a library of functions for operating on strings exists and can be accessed by using the header declaration `#include <string.h>`.
- It provides the following operations:
  - `strcmp(s, t)`: compares two strings `s` and `t`.
  - `strcat(s, t)`: concatenates string `t` to `s`.
  - `strlen(s)`: computes the length of string `s`.
  - `strcpy(s, t)`: copies string `t` into `s`.

String Operations

- Since strings are just arrays and are not treated in any special way, operations on strings are not provided in C.
- However, a library of functions for operating on strings exists and can be accessed by using the header declaration `#include <string.h>`.
- It provides the following operations:
  - `strcmp(s, t)`: compares two strings `s` and `t`.
  - `strcat(s, t)`: concatenates string `t` to `s`.
  - `strlen(s)`: computes the length of string `s`.
  - `strcpy(s, t)`: copies string `t` into `s`.

String Operations

- Since strings are just arrays and are not treated in any special way, operations on strings are not provided in C.
- However, a library of functions for operating on strings exists and can be accessed by using the header declaration `#include <string.h>`.
- It provides the following operations:
  - `strcmp(s, t)`: compares two strings `s` and `t`.
  - `strcat(s, t)`: concatenates string `t` to `s`.
  - `strlen(s)`: computes the length of string `s`.
  - `strcpy(s, t)`: copies string `t` into `s`.

String Operations

- Since strings are just arrays and are not treated in any special way, operations on strings are not provided in C.
- However, a library of functions for operating on strings exists and can be accessed by using the header declaration `#include <string.h>`.
- It provides the following operations:
  - `strcmp(s, t)`: compares two strings `s` and `t`.
  - `strcat(s, t)`: concatenates string `t` to `s`.
  - `strlen(s)`: computes the length of string `s`.
  - `strcpy(s, t)`: copies string `t` into `s`.

Implementing strcmp

```
/* Compares two input strings */
int my_strcmp(char s[], char t[])
{
    for (int i = 0; (s[i] != '\0') && (t[i] != '\0'); i++)
        if (s[i] != t[i]) /* strings not equal */
            break;

    return (int) (s[i] - t[i]);
}
```

Implementing strcmp: Alternative

```
/* Compares two input strings */
int my_strcmp(char *s, char *t)
{
    for (; (*s != '\0') && (*t != '\0'); s++, t++)
        if (*s != *t) /* strings not equal */
            break;

    return (int) (*s - *t);
}
```