ESC101N
Fundamentals of Computing

Arnab Bhattacharya
arnabb@iitk.ac.in

Indian Institute of Technology, Kanpur
http://www.iitk.ac.in/esc101/

1st semester, 2010-11

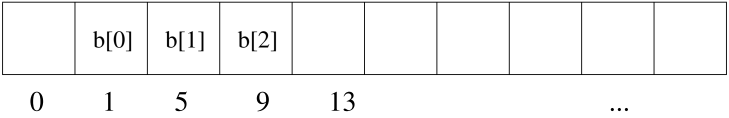
Tue, Wed, Fri 0800-0900 at L7

Multi-dimensional arrays

- A single-dimensional array

```
int b[3];
```

is stored in the following way:



b[0] b[1] b[2]

0 1 5 9 13 ...

Multi-dimensional arrays

- A single-dimensional array

```
int b[3];
```

is stored in the following way:

	b[0]	b[1]	b[2]						
0	1	5	9	13				...	

- Multi-dimensional arrays are stored row-wise in memory
- A two-dimensional array

```
int a[2][3];
```

is stored in the following way:

	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]			
0	1	5	9	13	17	21	25	...	

Multi-dimensional arrays and pointers

- A multi-dimensional array name is again a pointer
- An array element `a[i][j]` is equivalent to `*(*(a + i) + j)`
    - `a` points to first row
    - `(a + i)` points to *i*th row
    - `*(a + i)` points to first element of *i*th row
    - `(*(a + i) + j)` points to *j*th element of *i*th row
    - `*(*(a + i) + j)` is the value of *j*th element of *i*th row

Multi-dimensional arrays and pointers

- A multi-dimensional array name is again a pointer
- An array element `a[i][j]` is equivalent to `*(*(a + i) + j)`
  - `a` points to first row
  - `(a + i)` points to *i*th row
  - `*(a + i)` points to first element of *i*th row
  - `(*(a + i) + j)` points to *j*th element of *i*th row
  - `*(*(a + i) + j)` is the value of *j*th element of *i*th row
- If `a` is declared as

  ```c
  int a[2][3];
  ```

  address of `a[i][j]`

  $= a + i * sizeof(\text{row}) + j * sizeof(int)$

  $= a + i * (\text{number of elements in row}) * sizeof(int) + j * sizeof(int)$

  $= a + 12 * i + 4 * j$
- For example, if `a` is at address 1, `a[1][1]` is at address $1 + 12 * 1 + 4 * 1 = 17$

Multi-dimensional arrays and pointers

- A multi-dimensional array name is again a pointer
- An array element `a[i][j]` is equivalent to `*(*(a + i) + j)`
  - `a` points to first row
  - `(a + i)` points to *i*th row
  - `*(a + i)` points to first element of *i*th row
  - `(*(a + i) + j)` points to *j*th element of *i*th row
  - `*(*(a + i) + j)` is the value of *j*th element of *i*th row
- If `a` is declared as

  ```
  int a[2][3];
  ```

  address of `a[i][j]`

  $$= a + i * sizeof(\text{row}) + j * sizeof(int)$$

  $$= a + i * (\text{number of elements in row}) * sizeof(int) + j * sizeof(int)$$

  $$= a + 12 * i + 4 * j$$
- For example, if `a` is at address 1, `a[1][1]` is at address $1 + 12 * 1 + 4 * 1 = 17$
- Since size of row requires number of elements, this number is required when multi-dimensional arrays are passed to functions

Passing multi-dimensional arrays to functions

- Size of row is required when multi-dimensional arrays are passed to functions

```
void f(int b[][3])
{
   ...
}
```

- It is called by simply passing the array name

```
f(a);
```

- Reason: To compute the address of `a + 1` correctly
- `a + 1` points to the next *row*
- So, size of row, i.e., number of elements in the row is required

Passing multi-dimensional arrays to functions

- Size of row is required when multi-dimensional arrays are passed to functions

```
void f(int b[][3])
{
   ...
}
```

- It is called by simply passing the array name

```
f(a);
```

- Reason: To compute the address of `a + 1` correctly
- `a + 1` points to the next *row*
- So, size of row, i.e., number of elements in the row is required
- Passing an array with more or less number of elements in the row results in accessing elements row-wise

Passing multi-dimensional arrays to functions

- Size of row is required when multi-dimensional arrays are passed to functions

```
void f(int b[][3])
{
   ...
}
```

- It is called by simply passing the array name

```
f(a);
```

- Reason: To compute the address of `a + 1` correctly
- `a + 1` points to the next *row*
- So, size of row, i.e., number of elements in the row is required
- Passing an array with more or less number of elements in the row results in accessing elements row-wise
- For the same reason, for multi-dimensional arrays, all sizes except the first are required

Pointer operations I

```
#include <stdio.h>

void f_eq(int b[][4])
{
    int i, j;

    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 4; j++)
            printf("%d\t", b[i][j]);
        printf("\n");
    }
}

void f_more(int b[][5])
{
    int i, j;

    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 5; j++)
            printf("%d\t", b[i][j]);
        printf("\n");
    }
}

void f_less(int b[][3])
{
    int i, j;
```

Pointer operations II

```
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
            printf("%d\t", b[i][j]);
        printf("\n");
    }
}

int main()
{
    int a[3][4];
    int i, j;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 4; j++)
            a[i][j] = 10 * i + j;

    f_eq(a);
    f_more(a);
    f_less(a);
}
```

Returning pointers

- Since a pointer is a variable, a function can return a pointer
- The declaration

  ```
  char *f(char *s, char *t)
  ```

  declares `f` to be a function that returns a pointer to `char`

Returning pointers

- Since a pointer is a variable, a function can return a pointer
- The declaration

  ```
  char *f(char *s, char *t)
  ```

  declares `f` to be a function that returns a pointer to `char`
- Example: Function that returns the copy of a string

  ```
  char *strcpy(char *t)
  {
    char s[30]; // note array declaration
    for (; *s = *t; s++, t++)
      ;
    return s; // note return of array name as pointer
  }
  ```

Arrays of pointers

- Since a pointer is a variable, arrays of pointers can be declared
- The declaration

  ```
  char *a[3];
  ```

  declares a to be an array of 3 pointers to char
- a[i] is a pointer to char

Arrays of pointers

- Since a pointer is a variable, arrays of pointers can be declared
- The declaration

  ```
  char *a[3];
  ```

  declares a to be an array of 3 pointers to char
- a[i] is a pointer to char
- Common way to declare arrays of strings
- Very useful since the strings can be of variable size

  ```
  char *a[3] =
  {
    ``Kolkata'',
    ``Kanpur'',
    ``Hyderabad''
  };
  ```

Arrays of pointers

- Since a pointer is a variable, arrays of pointers can be declared
- The declaration

  ```
  char *a[3];
  ```

  declares a to be an array of 3 pointers to char
- a[i] is a pointer to char
- Common way to declare arrays of strings
- Very useful since the strings can be of variable size

  ```
  char *a[3] =
  {
    ‘‘Kolkata’’,
    ‘‘Kanpur’’,
    ‘‘Hyderabad’’
  };
  ```

- How to declare the size of the array pointed to by each a[i] otherwise?
- It requires dynamic memory allocation

Dynamic memory allocation

- Dynamic memory allocation is required when the programmer cannot determine in advance how much space will be required by the program
- Putting a large number as the maximum limit works, but it wastes a lot of memory
- Space is dynamically allocated using malloc
- It takes *size* in bytes as a parameter and returns `void *`, i.e., a pointer without a specific type

Dynamic memory allocation

- Dynamic memory allocation is required when the programmer cannot determine in advance how much space will be required by the program
- Putting a large number as the maximum limit works, but it wastes a lot of memory
- Space is dynamically allocated using malloc
- It takes *size* in bytes as a parameter and returns `void *`, i.e., a pointer without a specific type
- So, it requires explicit type casting
- Example: The following code

  ```
  int *a;
  a = (int *)malloc(5 * sizeof(int));
  ```

  is equivalent to declaring an array of 5 integers

  ```
  int a[5];
  ```

- If space cannot be allocated, *null* pointer is returned

Dynamic memory allocation

- Dynamic memory allocation is required when the programmer cannot determine in advance how much space will be required by the program
- Putting a large number as the maximum limit works, but it wastes a lot of memory
- Space is dynamically allocated using malloc
- It takes *size* in bytes as a parameter and returns `void *`, i.e., a pointer without a specific type
- So, it requires explicit type casting
- Example: The following code

  ```
  int *a;
  a = (int *)malloc(5 * sizeof(int));
  ```

  is equivalent to declaring an array of 5 integers

  ```
  int a[5];
  ```
- If space cannot be allocated, *null* pointer is returned
- Space should be freed after use using free
- It takes a pointer allocated using `malloc` as a parameter

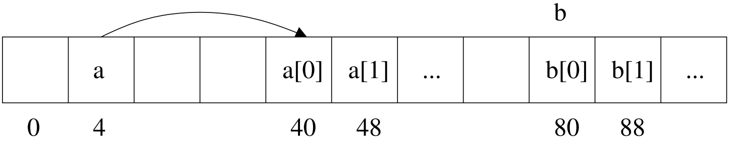
  ```
  free(a);
  ```

Dynamic array

```c
#include <stdio.h>
#include <stdlib.h> // required for malloc

int main()
{
    double *a;
    int i, n;
    double b[7];

    printf("Enter the size of array: ");
    scanf("%d", &n);

    a = (double *)malloc(n * sizeof(double));   // sizeof(double) is required as it is in
          bytes

    printf("Size of a is %d\n", sizeof(a)); // size of the pointer
    printf("Size of b is %d\n", sizeof(b)); // size of array is the total space allotted
          in bytes
    printf("Number of elements in b is %d\n", sizeof(b) / sizeof(double));

    for (i = 0; i < n; i++)
        a[i] = i;   // array notation

    for (i = 0; i < n; i++)
        printf("%lf %lf\t", a[i], *(a + i));
    printf("\n");

    printf("&a is %u, while a[0] is at %u\n", &a, &a[0]);   // a is a separate variable
          stored elsewhere
    printf("&b is %u, while b[0] is at %u\n", &b, &b[0]);   // b is not stored separately

    free(a);    // important as otherwise space is not freed
}
```

Dynamic allocation

- `a` is a pointer and is, therefore, stored as a variable separately
- When space is allocated, the value of `a` is appropriately modified to point to the beginning of that space
- `a[0]`, `a[1]`, etc. are stored in that space
- `b`, on the other hand, is just an array name and not stored explicitly



								b		
	a			a[0]	a[1]	...		b[0]	b[1]	...
0	4			40	48			80	88	

- `a` needs 4 bytes of storage as it is a pointer
- Each `a[i]` needs 8 bytes of storage as they are doubles

Arrays of pointers: dynamic allocation

- Since a pointer is a variable, arrays of pointers can be declared
- The declaration

  ```
  char *a[3];
  ```

  declares a to be an array of 3 pointers to char
- a[i] is a pointer to char

Arrays of pointers: dynamic allocation

- Since a pointer is a variable, arrays of pointers can be declared
- The declaration

  ```
  char *a[3];
  ```

  declares a to be an array of 3 pointers to char
- a[i] is a pointer to char
- Common way to declare arrays of strings
- Very useful since the strings can be of variable size
- How to declare the size of the array pointed to by each a[i]?
- It requires dynamic memory allocation

Dynamic array I

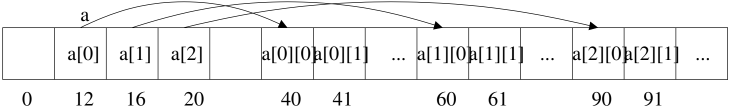
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *a[3];
    int i, n;

    for (i = 0; i < 3; i++)
    {
        printf("Enter maximum length of string %d: ", i);
        scanf("%d", &n);
        a[i] = (char *)malloc(n * sizeof(char));    // allocate space for each a[i]
    }

    for (i = 0; i < 3; i++)
    {
        printf("Enter string %d: ", i);
        scanf("%s", a[i]);
    }

    for (i = 0; i < 3; i++)
        printf("%s\n", a[i]);

    printf("Size of a is %d\n", sizeof(a));
    for (i = 0; i < 3; i++)
        printf("Size of a[%d] is %d\n", i, sizeof(a[i]));

    printf("a is at %u\n", &a);
```

Dynamic array II

```
    for (i = 0; i < 3; i++)
        printf("a[%d] is at %u\n", i, &a[i]);
    for (i = 0; i < 3; i++)
        printf("a[%d][0] is at %u\n", i, &a[i][0]);

    for (i = 0; i < 3; i++)
        free(a[i]); // free each a[i]
}
```

Array of pointers

- a is simply an array
- Each a[i] is a pointer and is, therefore, stored as a variable separately
- When space is allocated, the value of a[i] is appropriately modified to point to the beginning of that space
- a[i][0], a[i][1], etc. are stored in that space



- Each a[i] needs 4 bytes of storage as they are pointers
- Each a[i][j] needs 1 byte of storage as they are characters