Pointers and 2-D Arrays

Palash Dey
Department of Computer Science & Engg.
Indian Institute of Technology
Kharagpur

Slides credit: Prof. Indranil Sen Gupta

Concept of pointer to pointer

- A pointer stores the memory address of a variable.
- The pointer itself is a variable, and is stored in memory.
- We can define a pointer to pointer, to store the memory address of a pointer variable.

Example 1

```
#include <stdio.h>
main()
{
    int var;      int *ptr;      int **pptr;
   var = 3000;
   ptr = &var;        // Points to "var"
   pptr = &ptr;       // Points to "ptr"
   printf ("Value of var = %d \n", var );
   printf ("Value available at *ptr = %d \n", *ptr );
   printf ("Value available at **pptr = %d \n", **pptr);
}
```

Output

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

Example 2

```
#include <stdio.h>
main()
{
      int var;     int *ptr;     int **pptr;
      var = 3000;
      ptr = &var;
      pptr = &ptr;
      printf ("Address of var = %u \n", &var );
      printf ("Value of ptr = %u \n", ptr );
      printf ("Value stored at pptr = %u \n", *pptr);
}
```

Output

```
Address of var = 3974241144
Value of ptr = 3974241144
Value stored at pptr = 3974241144
```

What does array name mean in 2-D array?

```
int  a[10], b[5][3];
```

- We know that '**a**' is a constant pointer whose value is the address of the 0th element of the array **a[10]**.
- Similarly, **a+i** is the address of the ith element of the array.
- What is the meaning of '**b**' and what is its arithmetic?

How is a 2-D array is stored in memory?

- Starting from a given memory location, the elements are stored *row-wise* in consecutive memory locations.
  - x: starting address of the array in memory
  - c: number of columns
  - k: number of bytes allocated per array element

```
int b[5][3];
```

Element `b[i][j]` :: allocated memory location at address `x+(i*c+j)*k`

b[0][0] b[0][1] b[0]2] ….. b[1][0] b[1][1] b[1][2] …… b[2][0] b[2][1] b[2][2] ……..

Row 0 Row 1 Row 2

Arithmetic of 'b'

```
int b[5][3];
```

b[0][0] b[0][1] b[0]2] ….. b[1][0] b[1][1] b[1][2] …… b[2][0] b[2][1] b[2][2] ……..

Row 0 | Row 1 | Row 2

- `b` is the starting address of the 0th row
- `b+1` is the starting address of the 1th row
- `b+2` is the starting address of the 2th row
- In general, `b+i` represents the starting address of the ith row
- The size of a row will be: `c × sizeof(int)` bytes, where c is the number of columns.

Example 3

```
#include <stdio.h>
int main()
{
   int a[10], b[3][5];
   printf ("a:   %u \t   b:   %u \n", a, b);
   printf ("a+1: %u \t b+1: %u \n", a+1, b+1);
   printf ("a+2: %u \t b+2: %u \n", a+2, b+2);
   printf ("a+3: %u \t b+3: %u \n", a+3, b+3);
}
```

Output

```
a:   3217738332  b:   3217738272
a+1: 3217738336  b+1: 3217738292
a+2: 3217738340  b+2: 3217738312
a+3: 3217738344  b+3: 3217738332
```

Type of ‘b’

```
int b[3][5];
```

- ‘`b`’ is a pointer constant of type `int[][5]`, that is, a contiguous row of five integers.
- If such a pointer is incremented by one, it increases by `5×sizeof(int)` bytes.

Arithmetic of *(b+i)

- If ‘**b**’ is the address of the 0^{th} row, ***b** is the 0^{th} row itself.
  - A row may be viewed as a 1-D array, so ***b** is the starting address of this 1-D array, i.e. address of the 0^{th} element of the 0^{th} row.
- Similarly, `b+i` is the address of the i^{th} row, `*(b+i)` is the i^{th} row.
  - So `*(b+i)` is the address of the 0^{th} element of the i^{th} row.

For the array b[3][5]

- If *b is the address of the 0th element of the 0th row, *b+1 is the address of the 1th element of the 0th row.
- Similarly, *b+j is the address of the jth element of the 0th row.
- The difference between b+1 and b is 20 bytes, but the difference between *b+1 and *b is the `sizeof(int)`, that is, 4 bytes.

- So, `*(b+i)` is the address of the 0^{th} element of the i^{th} row.
- Thus, `*(b+i)+j` is the address of the j^{th} element of the i^{th} row.
  - That is, same as `&b[i][j]`.

```
*(b+i)+j is equivalent to &b[i][j]
```

Some Equivalences

`*(b + i) + j`	⟹	`&b[i][j]`
`*(*(b + i) + j)`	⟹	`b[i][j]`
`b[i] + j`	⟹	`&b[i][j]`
`*(b[i] + j)`	⟹	`b[i][j]`
`(*(b + i))[j]`	⟹	`b[i][j]`

Calculation of the address of `b[i][j]`

```
int  b[3][5]
```

- The C compiler can calculate the address of the j^{th} element of the i^{th} row using the following formula:

  ```
  b + k (5i + j)
  ```

  where `k = sizeof(int)`.

- The compiler needs the following:
  - Value of row and column indices
  - The number of columns
  - The size of the data type.

Passing 2-D Arrays to functions (recap)

1-D Array and Formal Parameter

- Consider the declaration: `int a[10];`
  - The array name ‘a’ is a constant pointer.
  - The formal parameter: `int x[]` or `int *x` is a pointer variable of the corresponding type, where the address of an array location is copied into the function.

```
void sort (int n, int x[]);

void sort (int n, int *x);
```

  - These two information are sufficient for the compiler to calculate the address of `x[i]`.

Formal parameter for 2-D Array

- Consider the declaration: `int b[ROW][COL];`
  - The C compiler needs the following information to calculate the address of `b[i][j]` (given i and j):
    - Starting address ‘`b`’
    - The data type of the array elements, that is, ‘`int`’
    - The number of columns ‘`COL`’
- Example:

```
void matadd (int row, int col, int a[][10],
                              int b[][10], int c[][10]);
```

An example

```
#include <stdio.h>

void transpose (int x[][3],
                      int n)
{
   int  p, q, t;

   for (p=0; p<n; p++)
     for (q=p; q<n; q++)
      {
         t = x[p][q];
         x[p][q] = x[q][p];
         x[q][p] = t;
      }
}
```

```
main()
{
   int a[3][3], p, q;

   for (p=0; p<3; p++)
     for (q=0; q<3; q++)
       scanf ("%d", &a[p][q]);
   transpose (a, 3);
   for (p=0; p<3; p++)
   {
     printf ("\n");
     for (q=0; q<3; q++)
        printf ("%d  ", a[p][q]);
   }
}
```

Dynamically Allocating 2-D Arrays

You may recall ….

- We have discussed earlier the issue of dynamically allocating space for 1-D arrays.
  - Using `malloc()` library function.
- Pros and cons of this approach:
  - The space gets allocated in global data area called heap (not on the stack), and hence does not evaporate at the end of function call.
  - The conventional method allocates space in the stack as part of the activation record, and so is not available across function calls.

Looking back at pointer arithmetic

```
int  *p, (*q)[5], *r[3], **s;
```

- Variable '`p`' can be used to point to an integer.

  Thus, `p+i` will mean: `p + i * sizeof(int)`

- Variable '`q`' can be used to point to an integer array of size 5.

  Hence, `q+i` will mean: `q + i*5*sizeof(int)`

- '`r`' is not a variable but a constant pointer (name of an array, each element of the array is an `int*`).

  So, `r+i` will mean: `r + i * sizeof(int*)`

- Variable '`s`' can be used to point to a location of type `int*` .

  Thus, `s+i` will mean: `s + i*sizeof(int*)`

Some typical values ….

`sizeof(int):`	4
`sizeof(int *):`	8
`sizeof(int [5]):`	20
`sizeof(int (*)[5]):`	8
`sizeof(int **):`	8

How was 1-D array dynamically allocated?

- Sample code segment:

```
int *p, n, i;
scanf ("%d", &n);
p = (int *) malloc (n * sizeof(int));
```

- Array elements can be accessed equivalently as:

```
p[i] = 20;
*(p+i) = 20;
```

Methods to allocate space for 2-D array

1. Variable number of rows, fixed number of columns
2. Variable number of columns, but fixed number of rows
3. Both number of rows and columns variable

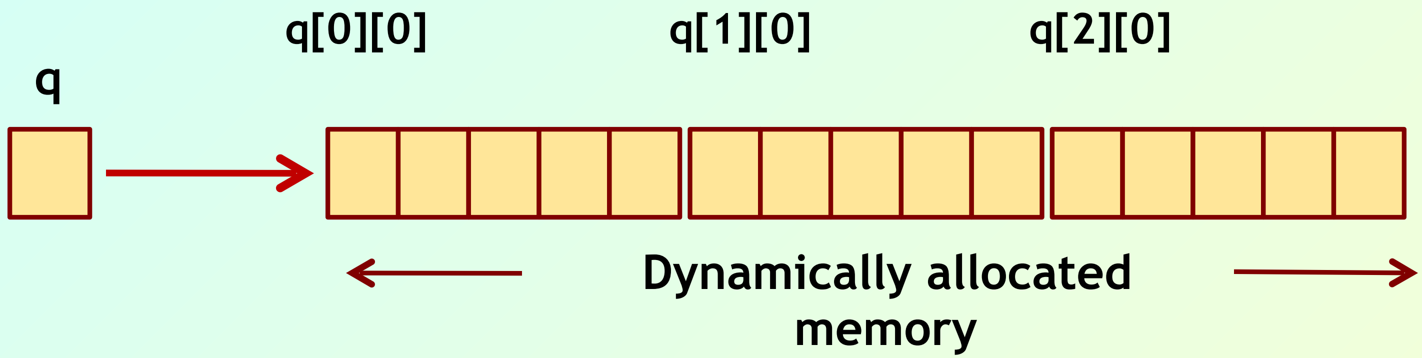
Dynamically Allocating 2-D Arrays

Variable number of rows
Fixed number of columns

1:: Allocating space for 2-D array *n×5*

- We can use a pointer of type `(*q)[5]` to allocate space for the array of n rows and 5 columns.

```
int  (*q)[5], n;
printf("Enter nos. of rows:");
scanf("%d", &n);
q = (int (*)[5]) malloc(n*5*sizeof(int));
```


q[0][0]
q[1][0]
q[2][0]
q
Dynamically allocated memory

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int (*q)[5],rows,i,j;
   printf("Enter the number of Rows: ") ;
     scanf("%d", &rows);

   q = (int (*)[5]) malloc (rows*5*sizeof(int));

   for(i=0; i<rows; ++i)
     for(j=0; j<5; ++j)
         q[i][j]=2*i+3*j;
   for(i=0; i<rows; ++i) {
     for(j=0; j<5; ++j)
         printf("%d ", q[i][j]);
     printf("\n");
   }
   return 0;
}
```

```
Enter the number of Rows: 3
0 3 6 9 12
2 5 8 11 14
4 7 10 13 16
```

- Some observations:
  - ‘`q`’ points to the 0th row of a 5-element array
  - ‘`q+i`’ points to the ith row of a 5-element array
  - `*q` is the address of `q[0][0]`, that is, `&q[0][0]`
  - `*q+j` is the address of `q[0][j]`, that is, `&q[0][j]`
  - `*(q+i)+j` is address of `q[i][j]`, that is, `&q[i][j]`
  - `**q` is `q[0][0]`
  - `*(*q+j)` is `q[0][j]`
  - `*(*(q+i)+j)` is `q[i][j]`

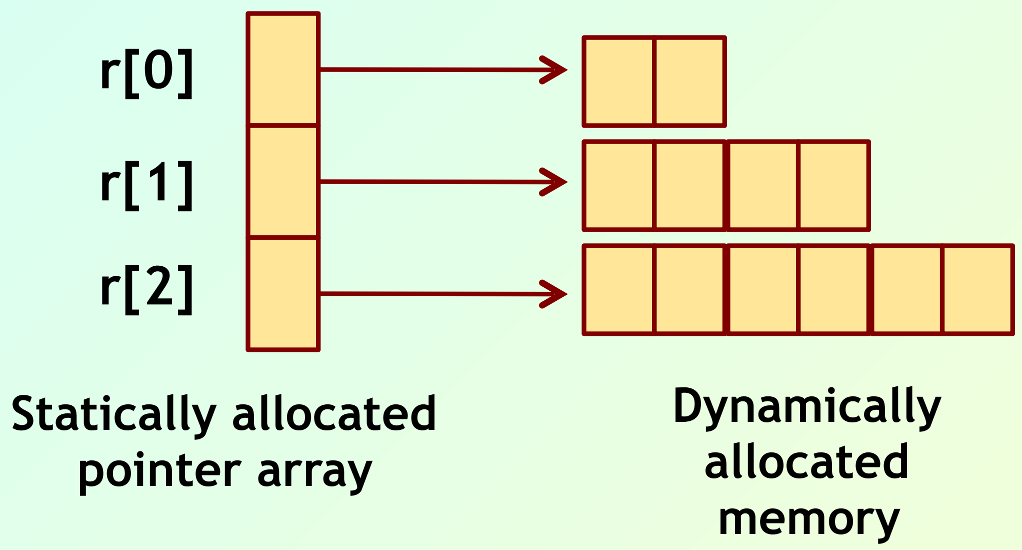
Dynamically Allocating 2-D Arrays

Fixed number of rows
Variable number of columns

2:: Allocating space for 2-D array $3 \times m$

- We can use a pointer array of size 3, where the **i**th element of the array will point to the **i**th row of length **m**.
  - Possible to have different number of elements in different rows.

```
int  *r[3], i, c;
printf("Enter nos. of columns:");
scanf("%d", &c);
for (i=0;i<3;i++)
   r[i] = (int *) malloc (c*sizeof(int));
```


r[0]
r[1]
r[2]
Statically allocated pointer array
Dynamically allocated memory

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *r[3], i, j, col;
  for(i=0; i<3; ++i) {
    col = 2 * (i+1);
    r[i] = (int *) malloc (col*sizeof(int));
    for(j=0; j<col; ++j)
      r[i][j] = i + j;
  }
 for(i=0; i<3; ++i) {
    col = 2 * (i+1);
    for(j=0; j<col; ++j)
       printf("%d ", r[i][j]);
    printf("\n");
  }
  return 0;
}
```

```
0 1
1 2 3 4
2 3 4 5 6 7
```

- Some observations:
  - `r[i]` is the `i`th pointer, which stores the address of the `0`th element of the `i`th row.
  - So, `r[i]+j` is the address of the `j`th element of the `i`th row.
  - `*(r[i]+j)`, same as `r[i][j]`, is the `j`th element of the `i`th row.

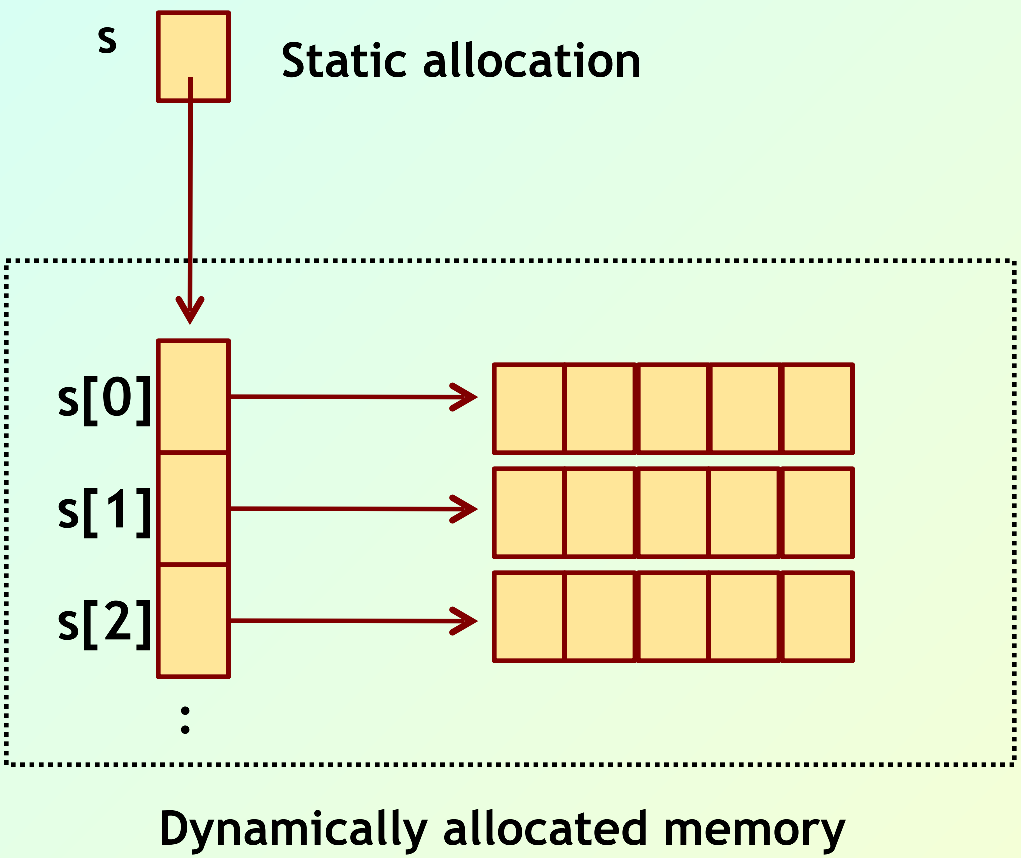
Dynamically Allocating 2-D Arrays

Both number of rows and columns are variable

3: Dynamic allocation of *r×c* array

- We can allocate a 2-D array of variable number of rows and columns, where both the number of rows and the number of columns as inputs.

```
int  **s, r, c;
printf("Enter nos. of rows, columns:");
scanf("%d %d", &r, &c);
s = (int **) malloc(r * sizeof(int *));
for (i=0;i<r;i++)
  s[i] = (int *) malloc(c * sizeof(int));
```


s
Static allocation
s[0]
s[1]
s[2]
:
Dynamically allocated memory

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int **s, row, column, i, j;
  printf("Enter Row & Column:\n”);
    scanf("%d %d", &row, &column);
  s = (int **) malloc(row*sizeof(int *));
  for(i=0; i<row; ++i) {
    s[i] = (int *) malloc(column*sizeof(int));
    for(j=0; j<column; ++j)
      s[i][j] = i+j ;
  }
 for(i=0; i<row; ++i) {
    for(j=0; j<column; ++j)
      printf("%d ", s[i][j]);
    printf("\n”);
  }
  return 0;
}
```

```
Enter Row and Column:
3 5
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
```

- Some observations:
  - `s+i` is the address of the `i`th element of the pointer array.
  - `*(s+i)`, which is the same as `s[i]`, is the `i`th element of the pointer array that stores the address of the `0`th element of the `i`th row.
  - `s[i]+j` is the address of the `j`th element of the `i`th row.
  - `*(s[i]+j)`, which is the same as `s[i][j]`, is the `j`th element of the `i`th row.

Example with 2-D Array

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

int **allocate (int h, int w)
    {
      int **p;
      int i, j;

      p = (int **) calloc (h, sizeof(int *) );
      for (i=0;i<h;i++)
        p[i] = (int *) calloc (w,sizeof(int));
      return(p);
    }
```

Allocate array of pointers

Allocate array of integers for each row

```c
void read_data (int **p, int h, int w)
  {
        int i, j;
        for (i=0;i<h;i++)
          for (j=0;j<w;j++)
            scanf ("%d", &p[i][j]);
  }

void print_data (int **p, int h, int w)
  {
       int i, j;
        for (i=0;i<h;i++)
        {
        for (j=0;j<w;j++)
          printf ("%5d ", p[i][j]);
         printf ("\n");
        }
}
```

Elements accessed like 2-D array elements.

```
main()
{
   int **p;
   int M, N;

   printf ("Give M and N \n");
   scanf ("%d%d", &M, &N);
   p = allocate (M, N);
   read_data (p, M, N);
   printf ("\nThe array read as \n");
   print_data (p, M, N);
}
```

```
Give M and N
3 3
1 2 3
4 5 6
7 8 9
The array read as
    1      2      3
    4      5      6
    7      8      9
```