2D Arrays and Double Pointers

Bryn Mawr College
CS246 Programming Paradigm

2D Arrays

- **int A[m][n];**
- The number of bytes: **m*n*sizeof(int).**

```
#define n 2
#define m 3
int A[n][m];

int A[2][3]={{1,2,3},{4,5,6}};
```

- For 1D array, to access array elements:
  - `A[i]`
  - `*(A+i)`

Access 2D Arrays Using Array Name

- **int A[m][n];**
- We can think of
  A[0] as the address of row 0,
  A[1] as the address of row 1
- In general: **A[i][j] = *(A[i] + j) = *(*(A+i)+j)**
- Example: **A[0][2] = *(A[0] + 2)**
  - Note that: **A[0] = *A**
- Hence, if **A** is a 2D **int** array, we can think of **A** as a pointer to a pointer to an integer. That is, **int****

Access 2D Arrays Using Array Name

- **int A[m][n];**
- A dereference of **A** : ***A**
  - the address of row 0 or **A[0]**
  - **A[0]** is an **int***
- A dereference of **A[0]** : ***A[0]**
  - the first element of row 0 or **A[0][0]**
  - ****A = A[0][0]** is an **int**

Array Equation

int A[4][3];

A00	A01	A02	A10	A11	A12	A20	A21	A22	A30	A31	A32
A==A[0]			A[1]			A[2]			A[3]		

For an int array **A[m][n]**:
address(A[i][j]) = address(A[0][0]) + (i × n + j) × size(int)

A[i] is equivalent to *(A+i)
&A[i][0] = &(*(A[i]+0)) = &*A[i] = A[i]

Types

- Different types:
  - **&A**: address of the entire array of arrays of ints, i.e int[m][n]
  - **&A[0]**: same as **A**, address of the first element, i.e., int[n]
  - **&A[0][0]**: address of the first element of the first element, i.e., int.
  - **A**: int (*)[n]
  - ***A**: int *
- An array is treated as a pointer that points to the first element of the array.
- 2D array is NOT equivalent to a double pointer!
- 2D array is "equivalent" to a "pointer to row".

Double Pointer and 2D Array

```
int A[m][n], *ptr1, **ptr2;          WRONG
ptr2 = &ptr1;
ptr1 = (int *)A;
```

- The information on the array "width" (n) is lost.
- A possible way to make a double pointer work with a 2D array notation:
  - use an auxiliary array of pointers,
  - each of them points to a row of the original matrix.

```
int A[m][n], *aux[m], **ptr2;
ptr2 = (int **)aux;
for (i = 0 ; i < m ; i++) aux[i] = (int *)A+ i * n;
```

Pointers as Arguments

- All arguments in C functions are passed by value.
- To change the value of a variable passed to a function, the variable's address must be given to the function.

```
int foo (int* ptr){
   ……
}
```

- The function **foo** can be called as **foo(&x)**.
- The function **foo** changes the value of **x** by dereferenceing **x**.

Pointers as Arguments

```
int allocate(int* A, int n){
   if ((A=malloc(n*sizeof(int))) != NULL)
      return 0;
   return 1;
}
int* ptr;
if (allocate(ptr,10)! = 1)
     do_something;
```

Passing a 2D Array to a Function

```
int main()
{
        int A[3][3],i,j;

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

        printf(" Initialized data to: ");
        for(i = 0 ; i < 3 ; i++) {
                printf("\n");
                for(j = 0 ; j < 3 ; j++)
                        printf("%4.2d", A[i][j]);
        }
        printf("\n");

        f1(A);
        f2(A);
        f3(A);
        f4(A);
        f5(A);
}
```

Passing a 2D Array to a Function

- Declare as matrix, explicitly specify second dimension
- You don't have to specify the first dimension!

```
void f1(int A[][3])  {
     int i, j;

     for(i = 0 ; i < 3 ; i++) {
          printf("\n");
          for(j = 0 ; j < 3 ; j++)
                   printf("%4.2d", A[i][j]);
     }
     printf("\n");
}
```

Passing a 2D Array to a Function

- A pointer to array, second dimension is explicitly specified

```
void f2(int (*A)[3])  {
     int i, j;

     for(i = 0 ; i < 3 ; i++) {
          printf("\n");
          for(j = 0 ; j < 3 ; j++)
                   printf("%4.2d", A[i][j]);
     }
     printf("\n");
}
```

Passing a 2D Array to a Function

- Using a single pointer, the array is "flattened"

```
void f3(int *A)   {
      int i, j;

      for(i = 0 ; i < 3 ; i++) {
            printf("\n");
            for(j = 0 ; j < 3 ; j++)
                  printf("%4.2d", *(A+ 3*i + j));
      }
      printf("\n");
}
```

Passing a 2D Array to a Function

- A double pointer, using an auxiliary array of pointers
- Add the dimensions to the formal argument list if you allocate "index" at run-time.

```
void f4(int **A)   {
      int i, j, *index[3];
      for (i = 0 ; i < 3 ; i++)
            index[i] = (int *)A + 3*i;
      for(i = 0 ; i < 3 ; i++) {
            printf("\n");
            for(j = 0 ; j < 3 ; j++)
                  printf("%4.2d", index[i][j]);
      }
      printf("\n");
}
```

Passing a 2D Array to a Function

- A single pointer, using an auxiliary array of pointers

```
void f5(int *A[3])   {
      int i, j, *index[3];
      for (i = 0 ; i < 3 ; i++)
            index[i] = (int *)A + 3*i;
      for(i = 0 ; i < 3 ; i++) {
            printf("\n");
            for(j = 0 ; j < 3 ; j++)
                  printf("%4.2d", index[i][j]);
      }
      printf("\n");
}
```

Protecting Pointers

```
int foo(const int* ptr){
   /* *ptr cannot be changed */
}

int foo(int* const ptr){
   /* ptr cannot be changed */
}

int foo(const int* const ptr){
   /* neither ptr nor *ptr cannot be changed */
}
```

Exercise

Write a function that

- takes
  - the name of a file (char*) that contains ints,
  - an array of ints
  - the address of a variable count
- reads the file into the array.

Assume that the array has enough space to hold the file. count should be updated to the number of entries in the file.

```
int foo(char* filename, int A[], int* countptr){
   FILE* fp=NULL;
   int num=0;
   if ((fp=fopen(filename, "r")) != NULL){
      while (fscanf(fp, "%d",&num)>0) {
         A[*countptr]= num;
         *countptr += 1;
      } return 0;
   } else
      return 1;
}
```

Consider the following declaration.
int matrix;**
Write a function matrixAllocate that

- takes two integers, m and n and
- allocate an m by n block of memory.

```
int matrixAllocate(int*** Mptr, int n, int m){
    *Mptr = (int**)malloc(m*sizeof(int*));
    int i=0;
    for (i=0; i<m; i++)
        (*Mptr)[i] = malloc(n*sizeof(int));
}
```