Computer Architecture (Practical Class)
Dynamic Memory Allocation - Part II

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2022/2023

Static Multidimensional Arrays

Concept

- In C, multidimensional arrays are implemented as unidimensional arrays with row-major ordering
- You can think of them as arrays of arrays

Example: `int md_array[5][2];`

- Declares an array of 5 one-dimensional arrays of 2 integers each
- The array occupies $5 \times 2 \times sizeof(int)$ bytes
- It can be statically initialized with the declaration:

```
int md_array[5][2] = {{1,2},{3,4},{5,6},{7,8},{9,10}};
```

Static Multidimensional Arrays

Accessing value in C

```c
int get_value(int md_array[][2], int i, int j){
  /* return md_array[i][j]; */
  return *(*(md_array + i) + j);
}
```

Accessing value in Assembly

```
# only one memory access to get m[i][j]
get_value:
  # m in %rdi, i in %esi, j in %edx

  shll $3, %esi                 # each line has 2 ints (8 bytes)
  addq %rsi, %rdi               # address of line i
  movl (%rdi,%rdx,4), %eax      # m[i][j]
  ret
```

Variable-size Multidimensional Arrays (1/4)

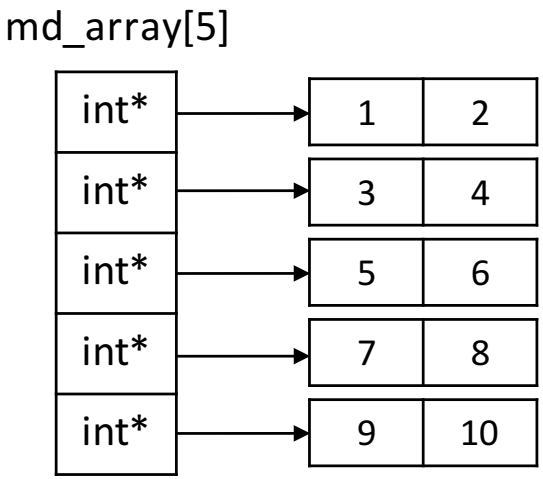
How can we dynamically allocate `md_array[Y][K]`?

- Variable `md_array` should be a dynamic array of pointers to int, with Y positions
- Each position of `md_array` will be initialized with a dynamic array of integers, with K positions

Variable-size Multidimensional Arrays (2/4)

In practice, what is variable `md_array` ?

- Variable `md_array` points to an array of pointers
- Each pointer in `md_array` points to an array of integers
- Thus, `md_array[y][k]` is equivalent to `*(md_array[y]+k)` and `*(*(md_array+y)+k)`



Variable-size Multidimensional Arrays (3/4)

Considering the previous example:

Expression	Type
$md\_array[2]$	Pointer to integer
md_array	Pointer to array of two integers
$md\_array + 1$	Pointer to array of two integers
$*(md\_array + 1)$	Pointer to integer
$*(md\_array + 2) + 1$	Pointer to integer
$*(*(md\_array + 2) + 1)$	Integer (md_array[2][1])
$*md\_array$	Pointer to integer
$**md\_array$	Integer (md_array[0][0])
$*(*md\_array + 1)$	Integer (md_array[0][1])

Variable-size Multidimensional Arrays (4/4)

Allocate variable-size multidimensional array

```
int main(void)
{
  int i, y=5,k=10; /* number of lines (Y) and columns (K) */
  int **a;         /* address of the multi-dimensional array */

  /*  array of int* with size Y */
  a = (int**) calloc(y,sizeof(int*));
  if(a == NULL){
    printf("Error reserving memory.\n"); exit(1);
  }
  for(i = 0; i < y ; i++){
    /* in each position of the pointer array,
       reserve memory for K integers */
    *(a+i) = (int*) calloc(k,sizeof(int)); //Note: *(a+i) same as a[i]
    if(a[i] == NULL){
      printf("Error reserving memory.\n"); exit(1);
    }
  }
  ... /* use multi-dimensional array */
  /* free memory */
  for(i = 0; i < y ; i++)
    free(*(a+i));
  free(a);
  return 0;
}
```

Variable-size Multidimensional Arrays

Accessing value in C

```c
int get_value(int **md_array, int i, int j){
  /* return md_array[i][j]; */
  return *(*(md_array+i) + j);
}
```

Accessing value in Assembly

```
# two memory accesses to get m[i][j]
get_value:
    # m in %rdi, i in %esi, j in %edx

    movq (%rdi,%rsi,8), %rdi      # address of line i
    movl (%rdi,%rdx,4), %eax      # m[i][j]
    ret
```

Observations

- Manipulating static and variable-size arrays is syntactically similar in C but the allocation mechanisms are completely different
- Declaring a multi-dimensional array `int a[5][2]`, statically reserves space for 10 integers in the stack, that can be readily accessed
- Declaring a dynamic multi-dimensional array can be done with `int **a`, where only a pointer to a pointer to `int` is declared
  - It must be properly initialized to a valid memory address in the heap that will be an array of pointers, and then each pointer in the array must be also initialized
- You can also mix the two concepts and declare a dynamic multi-dimensional array using a static array of pointers `int *a[5]` in the stack
  - Each of those pointers must be initialized to a valid memory address in the heap

Practice (1/2)

1. Create an array of strings to store the names of students of the same class. Consider that the number of students and the size of each name is variable, and unknown before runtime.
   - Read the number of students in the class first;
   - Read a string, and then copy it to the rightmost position in the array, reserving the necessary number of bytes.

2. Create a variable-size multidimensional array of integers of size $n \times m$, with the values of n and m chosen by the user. Assume that the array is initialized with random values.
   - Implement, in Assembly, the function `int get_value(int **matrix, int y, int k)` which should return the value at `matrix[y][k]`

Practice (2/2)

- Consider the folowing data type:

```
typedef struct {
  char  age;
  int   id_number;
  short grades[10];
  char  name[80];
  char  address[120];
} student_t;
```

1. Dynamically allocate an array of type `student_t` with a number of elements given by the user.
2. In Assembly, implement the following functions:
   - `int get_id_number(student_t *vec, int k)` which returns the `id_number` of the student at index k in the array
   - `void copy_grades(student_t *vec, int k, short *new_grades)` which copies the 10 grades from the array `new_grades` to the `grades` member of the student at index) k in the array.