CS1100 - Lecture 12

Instructor : Jasine Babu

Teaching Assistants : Nikhila K N, Veena Prabhakaran

In the last class, we were discussing two-dimensional arrays. We know that all elements of a two-dimensional array are stored in contiguous memory locations. Elements in Row `0` are stored first, followed by elements in Row `1` and so on. We also know that we can explicitly refer to the address of each row. This is one major advantage of having two-dimensional arrays, over single dimensional arrays. Any element of an array can be accessed by varying the index. In the case of two-dimensional arrays, we have two indices, each of which can be varied independently.

If we have an array of integers `int a[4][3]`, we have seen that `a[0], a[1], a[2] and a[3]` are themselves single dimensional arrays. Each of them is a single dimensional array of 3 integer values. For example, `a[1]` is a single dimensional array of integers, whose elements are `a[1][0], a[1][1], a[1][2]`. Just as the name of a single dimensional array represents the address of the first element of the array, `a[0], a[1], a[2], a[3]` also represent addresses. The expression `a[0]` represents the address of the first element in 0^{th} row, i.e., `&a[0][0]`. In general, the expression `a[i]` corresponds to `&a[i][0]`.

We have a related concept called array of pointers, which will be explained in the following section.

Array of pointers

We already know that the declaration `int *p;` means that `p` is a variable, which is a pointer to a memory location holding an integer value; i.e. the value of `p` is the address of some other integer variable. Similarly, we can declare an array of pointers, where the value of each array element is the address of an integer variable.

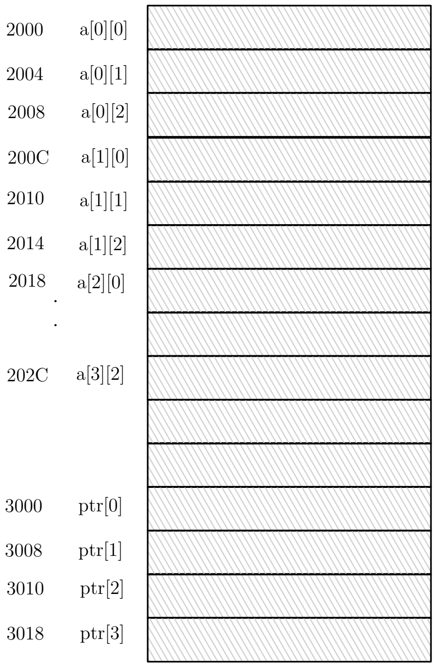
The declaration `int *ptr[4];` means, `ptr` is an array of 4 pointers, where each element of `ptr` can store the address of an integer variable. That is, the values of `ptr[0]`, `ptr[1]`, `ptr[2]`, `ptr[3]` are addresses of some locations storing integer values.

Suppose, we have an array of pointers declared as `int *ptr[4];` and a two-dimensional array declared as `int a[4][3];`. The instruction `ptr[0]=a[0]` is valid. Because `ptr[0]` can hold an address of an integer variable and `a[0]` denote `&a[0][0]`, which is the address of an inter variable. The value of `ptr[0]` can be changed by assigning new values, but the value of `a[0]` cannot be changed, because `a[0]` is not a variable.

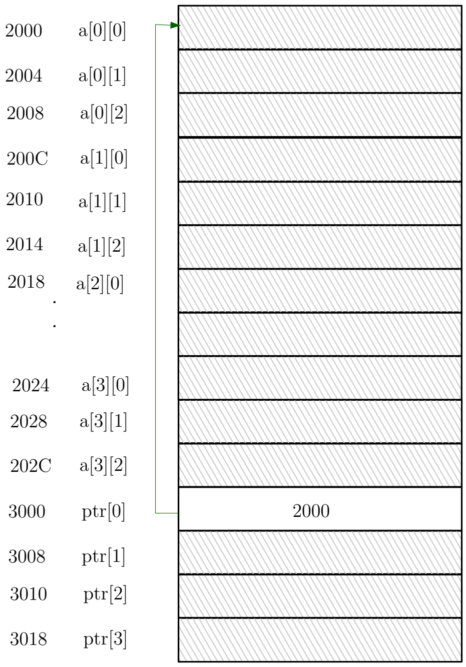
Consider the following code segment:

```
int *ptr[4],  a[4][3];

ptr[0]=a[0];
ptr[1]=a[1];
ptr[2]=a[2];
ptr[3]=a[3];
```

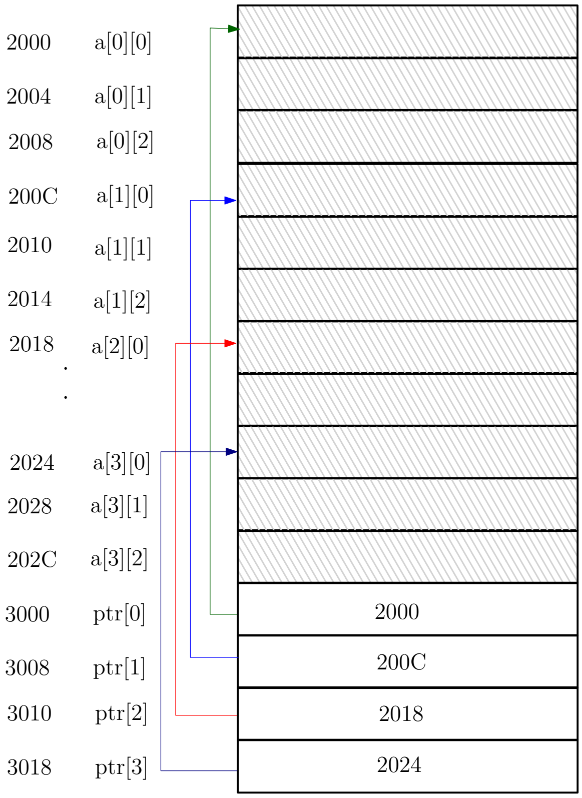
After the initial declarations, the contents of memory locations are as follows.



After executing the statement `ptr[0]=a[0]`, the address of the first element of Row 0 (i.e., `&a[0][0]`) is stored as the value of `ptr[0]`. The contents of memory locations after executing statement `ptr[0]=a[0]` are given in the following figure.



After executing the statement `ptr[1]=a[1]`, the value of `ptr[1]` becomes the address of the first element in Row 1, i.e. `&a[1][0]`. The following figure shows the contents of various memory locations after the execution of all the statements:



The following program gives an example of using an array of pointers to access the elements of a two-dimensional array.

```
#include<stdio.h>
int main()
{
  int i, j, *ptr1[3];
 /* ptr1 is an array of 3 pointers.
    The values of ptr1[0], ptr1[1], ptr1[2] are addresses. */
  int a[3][4]={{1, 2, 3, 4},
               {5, 6, 7, 8},
               {9, 10, 11, 12}};

  for(i=0; i<3; i++)
  {
    ptr1[i]=a[i];
  }

  for(i=0; i<3; i++)
  {
     for(j=0; j<4; j++)
     {
        printf("%2d ", *(ptr1[i]+j) );
     }
```

```
      printf("\n");
   }
   return 0;
}
```

As we discussed, when the instruction `ptr1[i]=a[i];` is executed, the value of `ptr1[i]` becomes the address of the first element of row `i`, i.e., `&a[i][0]`. This instruction is executed in a loop as follows:

```
   for(i=0; i<3; i++)
   {
     ptr1[i]=a[i];
   }
```

After executing the above loop, for $i = 0, 1, 2$, the i^{th} element of the array `ptr1` stores the address of the first element of row `i` of `a`, i.e, `&a[i][0]`.

At this stage, the expression `ptr1[i]+j` has the same value as `&a[i][0] + j` × `number of locations required to store one integer`. This is same as the address of `&a[i][j]`. Therefore, `*(ptr1[i]+j)` is the same as `a[i][j]`.

From the above explanation, it is now clear that the following code will display the elements of the two- dimensional array `a` in the form of a `3` × `4` matrix.

```
for(i=0; i<3; i++)
{
     for(j=0; j<4; j++)
     {
        printf("%2d ", *(ptr1[i]+j) );
     }

     printf("\n");
}
```

After executing the above parts of the code, the matrix will be displayed as follows.

```
1  2  3  4
5  6  7  8
9 10 11 12
```

Pointer to an array

In this section, we will look at the concept of a pointer to an array of integers. If we have a declaration `int (*ptr1)[4];`, then `ptr1` is a pointer to an array of four integers. This means, the value of `ptr1` will be an address (i.e., the starting address) of an array containing four integers. For example, if we also have another declaration `int b[4];`, then the instruction `ptr1 = &b;` is valid. After executing this instruction, the starting address of the array `b` will become the value of the pointer variable `ptr1`.

Arithmetic operations on a pointer to an array

If `ptr1` is declared as above, and we have an instruction `ptr1 = ptr1 + 1;`, then the new value of `ptr1` becomes `old value of ptr1 + number of locations required to store an array of 4 integers`. This new value is equal to `old value of ptr1 + 4` × `number of locations required to store an integer`. Similarly, other arithmetic operations can also be done on a pointer to an array.

Example of using a pointer to an array of integers

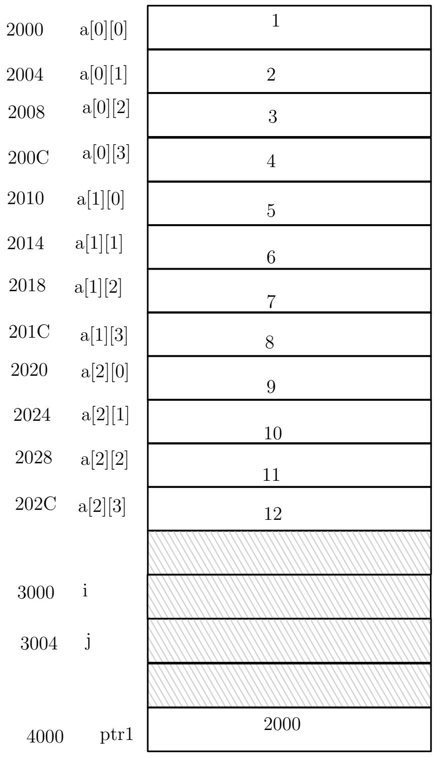
The following example shows accessing the elements of a two-dimensional array of integers using a pointer to an array.

```
#include<stdio.h>
int main()
{
  int i, j, (*ptr1)[4];
 //ptr1 is a pointer to an array of 4 integers.
  int a[3][4]={{1, 2, 3, 4},
               {5, 6, 7, 8},
               {9, 10, 11, 12}};

  ptr1 = &a[0];
  for(i=0; i<3; i++)
  {

     for(j=0; j<4; j++)
     {
        printf("%2d ", *(*ptr1+j) );
     }

     printf("\n");
     ptr1 = ptr1 + 1;
  }
  return 0;
}
```

Suppose the array elements are stored from memory address 2000 onwards. After the array initialization, the contents of memory locations are as given in the figure below:

2000	a[0][0]	1
2004	a[0][1]	2
2008	a[0][2]	3
200C	a[0][3]	4
2010	a[1][0]	5
2014	a[1][1]	6
2018	a[1][2]	7
201C	a[1][3]	8
2020	a[2][0]	9
2024	a[2][1]	10
2028	a[2][2]	11
202C	a[2][3]	12
		▨
3000	i	▨
3004	j	▨
		▨
4000	ptr1	▨

When the statement `ptr1 = &a[0];` is executed, the value of `ptr1` gets updated to `2000`, which is the address of `a[0]`. After executing this step, the contents of memory locations are as shown in the figure below.



Consider the point of execution when the control reaches the line `printf("%2d ", *(*ptr1+j) );` for the first time. Since `ptr1` has value `&a[0]`, the expression `*ptr1` stands for `a[0]`, which is same as `&a[0][0]`. Therefore, `*ptr1+j` stands for `&a[0][j]` and `*(*ptr1+j)` stands for `a[0][j]`. For the first time when the line is executed, the values of `j` is zero and so, the value of `a[0][0]` will be printed.

Now, consider the inner loop:

```
for(j=0; j<4; j++)
{
   printf("%2d ", *(*ptr1+j) );
}
```

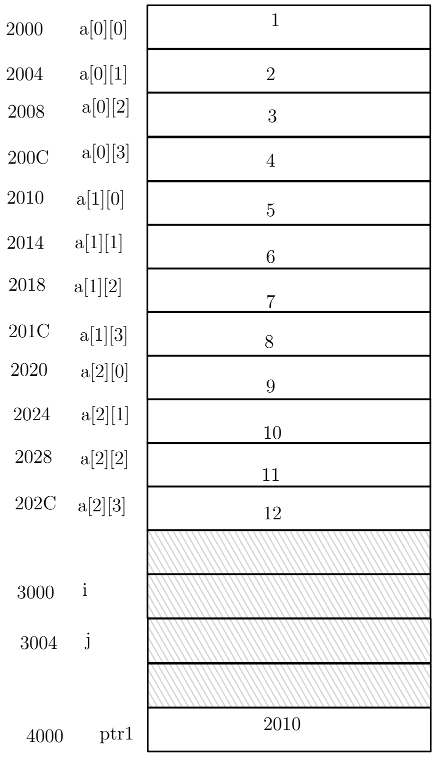
While executing this loop, the value of j changes from `0` to `3`. Since, `*(*ptr1+j)` stands for `a[0][j]`, when j sages from `0` to `3` in this loop, the values of `a[0][0], a[0][1], a[0][2], a[0][3]` gets printed.

Therefore, when the program control comes out of the above loop for the first time, the following is displayed on the screen.

```
 1  2  3  4
```

Recall that, at this point, the value of `ptr1` is `2000`, which is the address of `a[0]`. When the line `ptr1 = ptr1 + 1;` is executed during the first iteration of the outer `for`-loop, the value of `ptr1` becomes `2000 + number of locations required to store an array` `of 4 integers`. As per our figure, number of locations are required to store an array of 4 integers is 16. Therefore, the new value of `ptr1` will be `2000 + 16`. Since we have expressed addresses in hexadecimal, the new value of `ptr1` will be `2010`, which is `&a[1]` (note that `2000 + 16 = 2010` in hexadecimal).

After executing the instruction `ptr1 = ptr1 + 1;` in the first iteration of the outer `for`-loop, the contents of memory locations are as shown in the figure below.



Now, the second iteration of the outer `for`-loop begins. Once again, the following inner loop is executed:

```
for(j=0; j<4; j++)
{
   printf("%2d ", *(*ptr1+j) );
}
```

Since `ptr1` has value `&a[1]`, `*ptr1` is `a[1]`, which is same as `&a[1][0]`. Therefore, `(*ptr1+j)` is same as `&a[1][j]` and `*(*ptr1+j)` stands for `a[1][j]`. While executing this loop, the value of j changes from `0` to `3`. Since, `*(*ptr1+j)` stands for `a[1][j]`, when j changes from `0` to `3` in this loop, the values of `a[1][0], a[1][1], a[1][2], a[1][3]` gets printed.

Therefore, when the program control comes out of the above loop for the second time, elements of the second row of the two-dimensional array gets printed. The execution proceeds in a similar way, and the final output printed will be:

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
```

Some comments about compatibility of types

Suppose we have the following instructions in a program.

```
int *ptr[3], a[3][4], (*ptr1)[4];
ptr[0] = a[1];
ptr1 = &a[1];
```

From our discussions, it can be seen that the above set of instructions are valid: If we compile the above code, it will compile without any warnings.

However, note that compiling the following program will cause some compiler generated warnings about both the assignment instructions in the program.

```
int *ptr[3], a[3][4], (*ptr1)[4];
ptr[0] = &a[1];
ptr1 = a[1];
```

Let us consider the instruction `ptr[0] = &a[1];`. Here, since `ptr[0]` is a pointer to an integer, the address to be stored as the value `ptr[0]` has to be the address of an integer. However, the right hand side of the instruction is `&a[1]`. Since `a[1]` is an array of `4` integers, `&a[1]` is the address of an array of `4` integers. This is different **in type** from an address of an integer. This is the reason for the compiler warning generated about the instruction `ptr[0] = &a[1];`

Similarly, in the instruction `ptr1 = a[1];`, the variable on the left side is a pointer to an array of `4` integers, the value assigned to `ptr1` should have been the address of an array of `4` integers. However, the right hand side of the instruction is `a[1]` (or `&a[1][0]`), which is the address of an integer. Since `a[1]` represents the address of an integer, its type is different from an address of an array of integers. This is the reason for the compiler warning generated about the instruction `ptr1 = a[1];`

Matrix multiplication

A matrix of order $m \times n$ has m rows and n columns. If we want to multiply two matrices, the number of columns in the first matrix should be equal to the number of rows in the second matrix. Otherwise, multiplication is not defined. Suppose we have a 3×4 matrix

$$\text{A=} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 8 & 2 & 8 & 7 \\ 3 & 5 & 1 & 3 \end{bmatrix}$$

and another matrix B of order 4×2.

$$\text{B=} \begin{bmatrix} 5 & 8 \\ 2 & 3 \\ 1 & 7 \\ 4 & 2 \end{bmatrix}$$

The product matrix C = A × B of order 3×2 obtained by multiplying matrix A and B is of the form

$$\text{C=}\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \\ C_{20} & C_{21} \end{bmatrix}$$

Consider an element C_{10} in the matrix C. For calculating C_{10}, elements of the row 1 of matrix A are multiplied with the corresponding element in the 0^{th} column of B; and the products obtained are summed up together.

$$\text{A=}\begin{bmatrix} 1 & 2 & 3 & 4 \\ 8 & 2 & 8 & 7 \\ 3 & 5 & 1 & 3 \end{bmatrix} \qquad \text{B=}\begin{bmatrix} 5 & 8 \\ 2 & 3 \\ 1 & 7 \\ 4 & 2 \end{bmatrix}$$

$$C_{10} = 8 * 5 + 2 * 2 + 8 * 1 + 7 * 4$$

The expression $C_{10} = A_{10} * B_{00} + A_{11} * B_{10} + A_{12} * B_{20} + A_{13} * B_{30}$ calculates the value of the element C_{10}. In general, if A is a matrix of order $m \times n$ and B is a matrix of order $n \times p$ and C = A × B, then following expression calculate the value of C_{ij}:

$$C_{ij} = A_{i0} * B_{0j} + A_{i1} * B_{1j} + A_{i2} * B_{2j} + A_{i3} * B_{3j} + ... + A_{i(n-1)} * B_{(n-1)j}$$

Note that, each term in the summation on RHS have `i` as the row index of the element of `A` and `j` as the column index of the element of `B`. Moreover, from term to term, the column index of the element of `A` and the row index of the element of `B` vary from 0 to $n-1$.

We can use the following code to compute the value of C_{ij}:

```
result=0;
for(k=0; k<n; k++)
{
    result= result+a[i][k]*b[k][j];
}
```

After executing the above code, the variable `result` will store the value of C_{ij}.

The following program is based on the idea developed above. This program illustrates taking a row number `row` and a column number `col` as input from the user and computing the value of the element with index `[row][col]` in the product of two matrices defined in the program.

```
#include<stdio.h>
int main()
{
     int a[3][4] ={{1, 2, 4, 5},
                   {2, 3, 1, 7},
                   {4, 2, 1, 3}};
     int b[4][3] ={{5, 10, 15},
                   {2, 5, 7},
                   {4, 2, 6},
                   {9, 4, 3}};
     int result, i, j,row,col, r1, c1, r2, c2, k;
     r1=3;
     c1=4;
     r2=4;
     c2=3;
     printf("Matrix 1\n");
     for(i=0; i<r1; i++)
     {
        for(j=0; j<c1; j++)
        {
           printf("%2d ", a[i][j]);
        }
        printf("\n");
      }

      printf("Matrix 2\n");
      for(i=0; i<r2; i++)
      {
         for(j=0; j<c2; j++)
         {
            printf("%2d ", b[i][j]);
         }
          printf("\n");
      }
      printf("Give row and column of the element to display from product matrix \n");
      printf("(assume index start from 0)\n");
      scanf("%d%d",&row,&col);
      if(row>=0 && row<r1 && col>=0 && col<c2 )
      {
         result=0;
         for(k=0; k<c1; k++)
         {
            result= result+a[row][k]*b[k][col];
         }

          printf("result is %d\n", result);
      }
      else
          printf("index out of range\n");

}
```