CEng 140

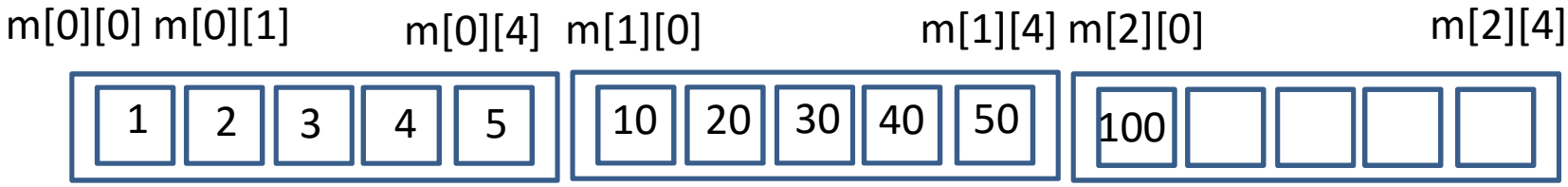
Multi Dimensional Arrays and Pointers

Multi-dimensional Arrays: Storage

- A multi-dim array in C is really a one-dim array [a contiguous area],
  - whose elements are themselves arrays (i.e., **arrays of arrays**)
  - and stored such that the **last subscript varies most rapidly** (i.e., row-order storage)
- Name of the multi-dim array is a **pointer to the first array**!

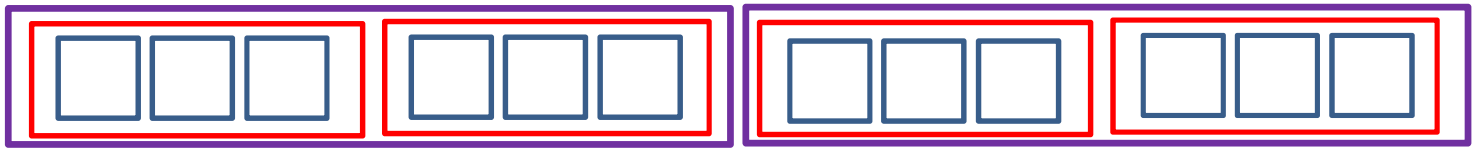
- A multi-dim array in C is really a one-dim array [a contiguous area],
  - whose elements are themselves arrays (i.e., **arrays of arrays**)
  - and stored such that the **last subscript varies most rapidly** (i.e., row-order storage)

```
int m[3][5] = {{1,2,3,4,5},
               {10,20, 30, 40,50},
               {100, 200, 300, 400, 500}};
```



- A multi-dim array in C is really a one-dim array [a contiguous area],
  - whose elements are themselves arrays (i.e., **arrays of arrays**)
  - and stored such that the **last subscript varies most rapidly** (i.e., row-order storage)

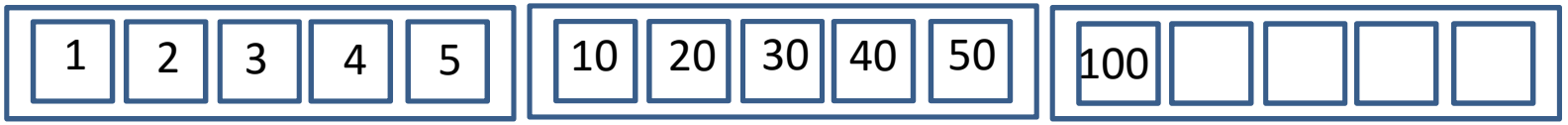
```
int m[2][2][3] = {
        { {1, 2, 3},   {2, 3, 4}},
        { {5, 6, 3},   {7, 8, 4}}  };
```



m[0][0][0] m[0][0][1] m[0][0][2] m[0][1][0] m[0][1][1] m[0][1][2]
m[1][0][0] m[1][0][1] m[1][0][2] m[1][1][0] m[1][1][1] m[1][1][2]

```
int m[3][5] = {{1,2,3,4,5},
               {10,20, 30, 40,50},
               {100, 200, 300, 400, 500}};
```

What does m[i][j] really mean?



m is a pointer to the first array
(i.e., a ptr to array of 5 elements)

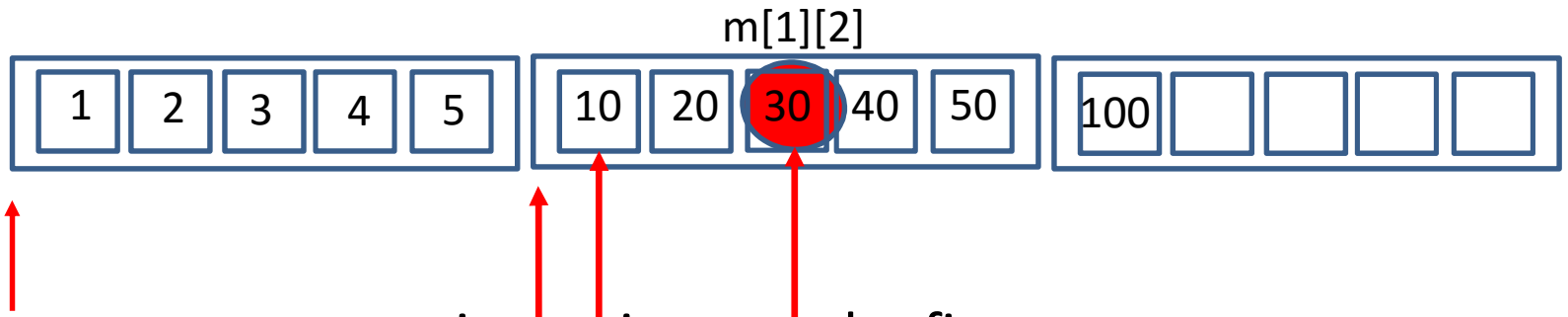
m+i is a pointer to the i-th array

*(m+i) is the i-th array pointer converted into a
pointer to the first element in the i-th array

*(m+i) + j is a pointer to the j-th element in the i-th array

* (*(m+i) +j) is m[i][j]

m[1][2]



m is a pointer to the first array
(i.e., a ptr to array of 5 elements)

m+1 is a pointer to the 1-th array

*(m+1) is the 1-th array pointer converted into a pointer to the first element in the 1-th array

*(m+1) + 2 is a pointer to the 2-th element in the 1-th array

* (*(m+1) +2) is m[1][2]

Imagine what happens for a 3 dim array!

A 2D array

int main()

{ int i, j, scores[3][5] = {1, 2, …,15};

What is the value of scores?

What is the type of scores?

What is the value of scores+1?

What is the type of scores+1?

What is the value of *(scores+1)?

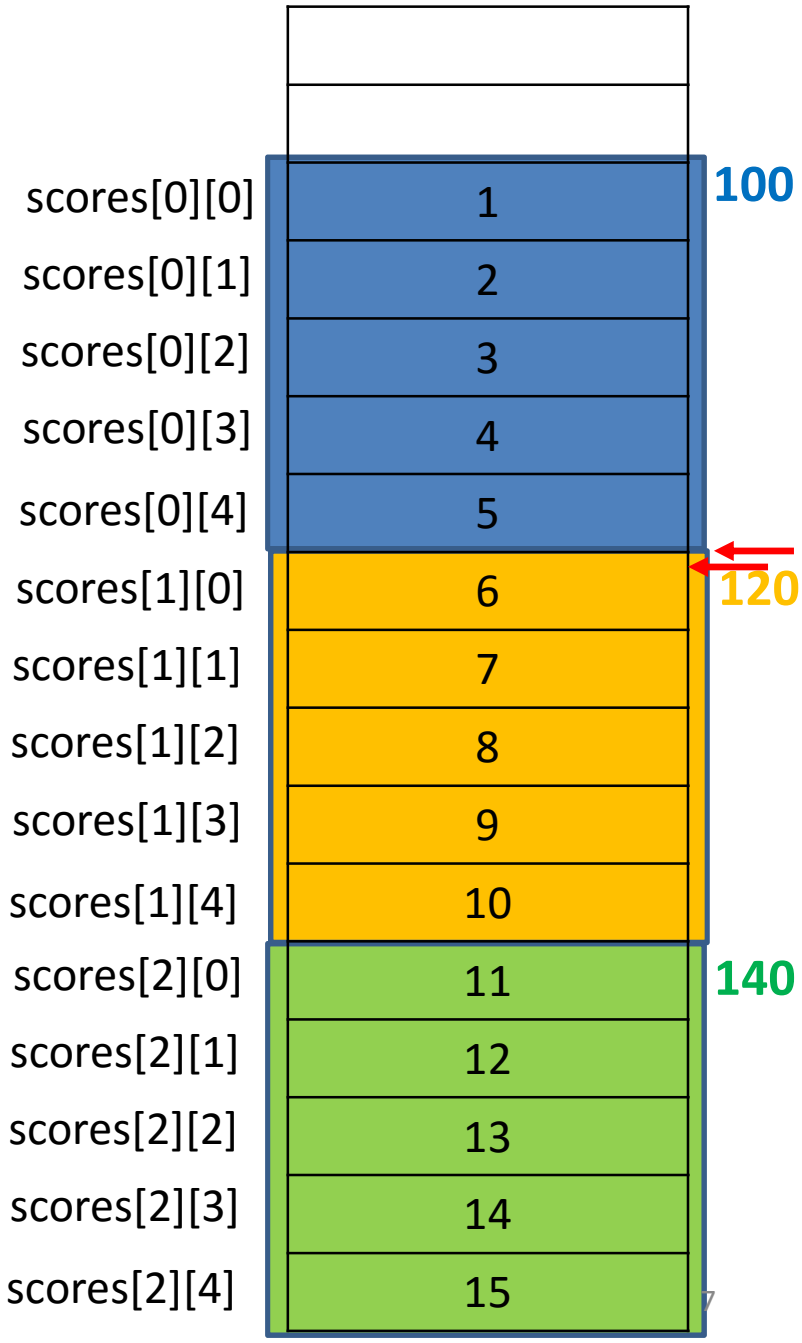
What is the type of *(scores+1)?

What is the value of *(scores+1)+1?

What is the type of *(scores+1)+1?

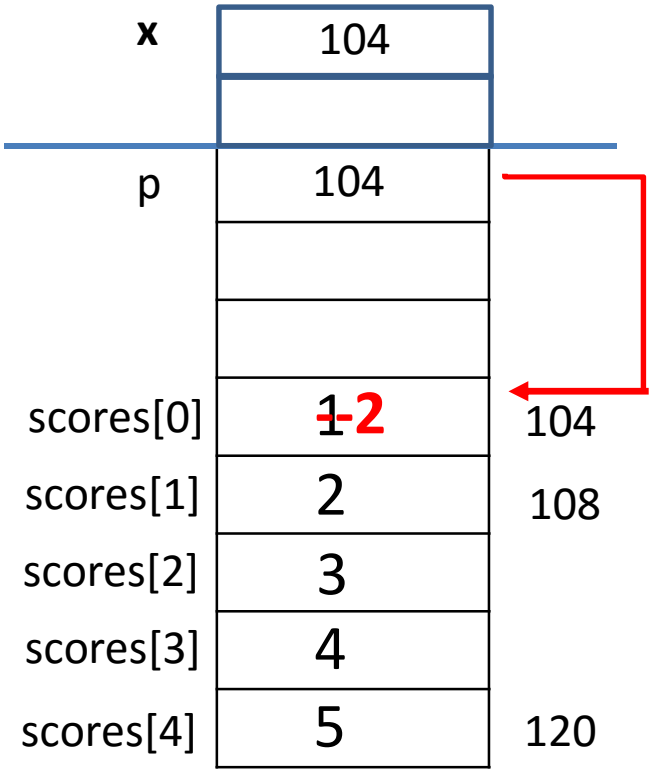
What is the value of *(*(scores+1)+1)?

What is the type of *(*(scores+1)+1)?



Accessing 1D array via a pointer

```
int main()
{ int scores[5] = {1,2,3,4,5};
   int *p, i;
   p=scores;
   for (i=0; i<5; i++)
        *(p+i) *= 2;   }
```



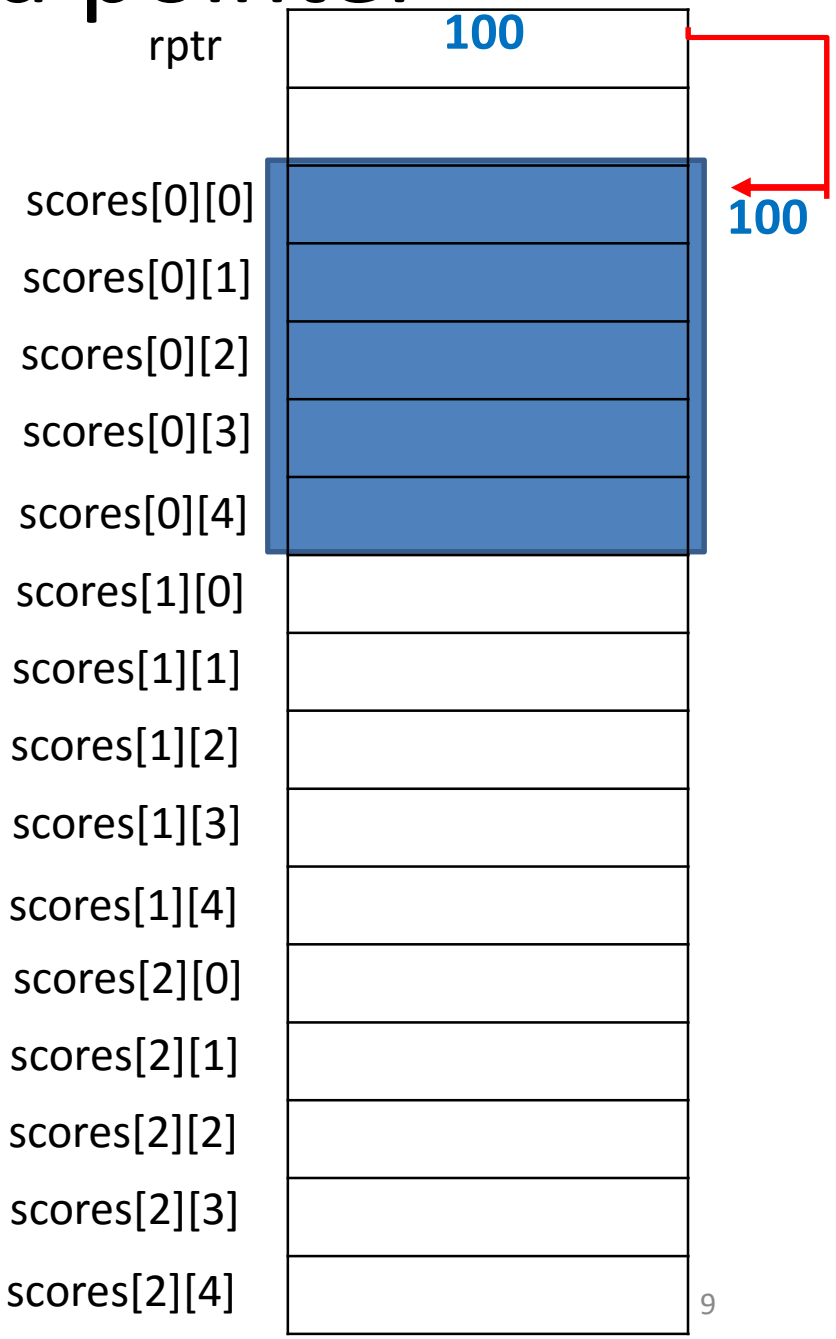

```
            int *x
  void f(int x[],int len)
  { int i;
   for (i=0; i<len; i++)
         *(x+i) *= 2;  /* x[i] *= 2; */ } }
```

CALL (in main here):

f(scores, 5);

Accessing 2D array via a pointer

```
int main()
{ int i, j, scores[3][5] = {…};
   /* Declare a pointer to the
        rows; i.e, array of 5 ints */
   int  (*rptr)[5];
   rptr=scores;
   for (i=0; i<3; i++)
      for (j=0; j<5; j++)
         *(*(rptr+i) + j) *= 2;   }
```



Accessing 2D array via a pointer

```
int main()
{ int i, j, scores[3][5] = {…};
  /* Declare a pointer to the
       rows; i.e, array of 5 ints */
  int  (*rptr)[5];
  rptr=scores;
  for (i=0; i<3; i++)
     for (j=0; j<5; j++)
       *(*(rptr+i) + j) *= 2;   }
```

Parameter is pointer to first array element, i.e., pointer to an array of 5 integers!

```
int (*x)[5]
void f(int x[][5],int nr, int nc)
{ int  (*rptr)[5];
  rptr=x;
  for (i=0; i<nr; i++)
     for (j=0; j<nc; j++)
       *(*(rptr+i) + j) *= 2;}
```

CALL (in main here):

```
f(scores, 3, 5);
```

Mystery solved!

- Recall from the previous weeks:

"When we declare a **multi-dim array** as a **parameter**, we must still specify **all** but the first dimension!"

- This is needed, so that when compiler sees m[i][j], it can compute the pointer arithmetics for **m+i**; i.e., it will go **i** "arrays" ahead from the base address **m**.
- Of course, you should still separately pass as paramaters the length of array for each dimension, to know the array boundaries.

Accessing 2D array via a pointer

int **(*x)[5]**

```
void f(int x[][5],int nr, int nc)
{ int  (*rptr)[5];
   rptr=x;
   for (i=0; i<nr; i++)
      for (j=0; j<nc; j++)
         *(*(rptr+i) + j) *= 2; OR
          rptr[i][j] *= 2; OR
         *(rptr[i]+ j) *= 2; OR
         (*(rptr+i))[j] *= 2;
```

Reminder

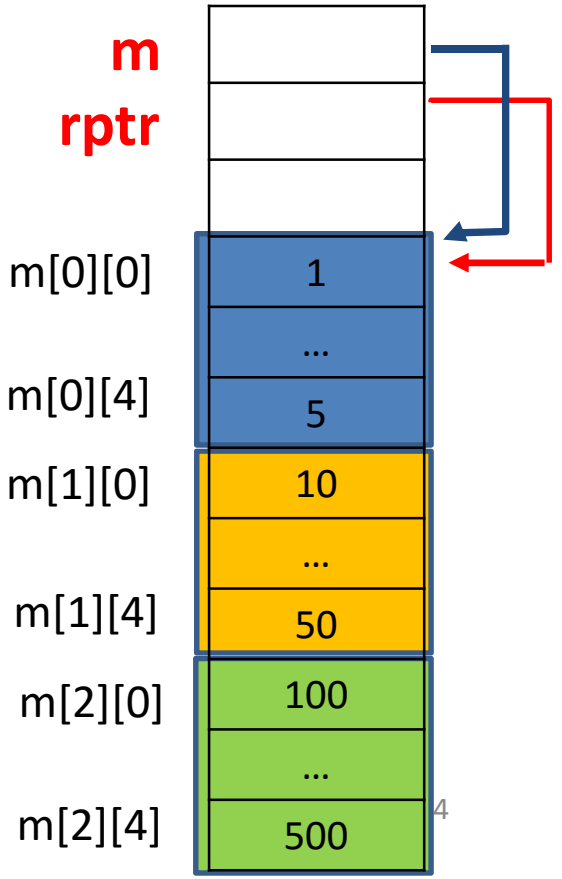
Operator	Type	Associativity
Fucntion call: () Array subscript: []		Left to right
(type) + - ++ -- **! & * sizeof**	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
< <= > >=	Binary	Left to right
== !=	Binary	Left to right
&&	Binary	Left to right
||	Binary	Left to right
= *= /= %= += -=	Binary	Right to left
,		Left to right

Example: Column Sum

```
void f(int m[][5],int no_rows, int no_cols, int target)
{ int  i, (*rptr)[5] = m, sum =0;
  for (i=0; i<no_rows; i++)
        sum +=  rptr[i][target]; OR
        sum +=  *(rptr[i]+ target); OR
        sum += *(*(rptr+i) + target); OR
        sum +=    (*(rptr+i))[target]; OR

       sum +=  (*rptr)[target];
       rptr++; OR
       sum +=  (*rptr++)[target];
```

```
int m[3][5] ={
{1,    2,    3,    4,    5},
{10,  20,   30,   40,   50},
{100, 200, 300, 400, 500}};
f(m, 3, 5, 1);
```



CEng 140

Multi Dimensional Arrays and Pointers

Dynamic 2D Arrays

Creating Dynamic 2D Arrays

- True 2D:
  - Both dimension lengths are known at compile time
  - Ex: I have 20 students and 5 int grades per student:
  - int stu_grades[20][5];
- Dynamic:
  - First dimension length is determined dynamically
  - Second dimension length is determined dynamically
  - Both dimension lengths are determined dynamically

Case 1: First dim dynamic

- I have 5 int grades per student, but number of students wil be determined during run-time

```
int main()
{  int (*stu_grades)[5]; /* ptr to a block of 5 ints */
   int no_of_stu, i, j, temp;
   scanf("%d", &no_of_stu);
   stu_grades = (int (*)[5]) malloc(sizeof(int [5])*no_of_stu);
   for (i=0; i< no_of_stu; i++)
     for (j=0; j<5; j++)
     {   scanf("%d", &temp);
         stu_grades[i][j] = temp; }
}        /* Lets draw this on board */
```

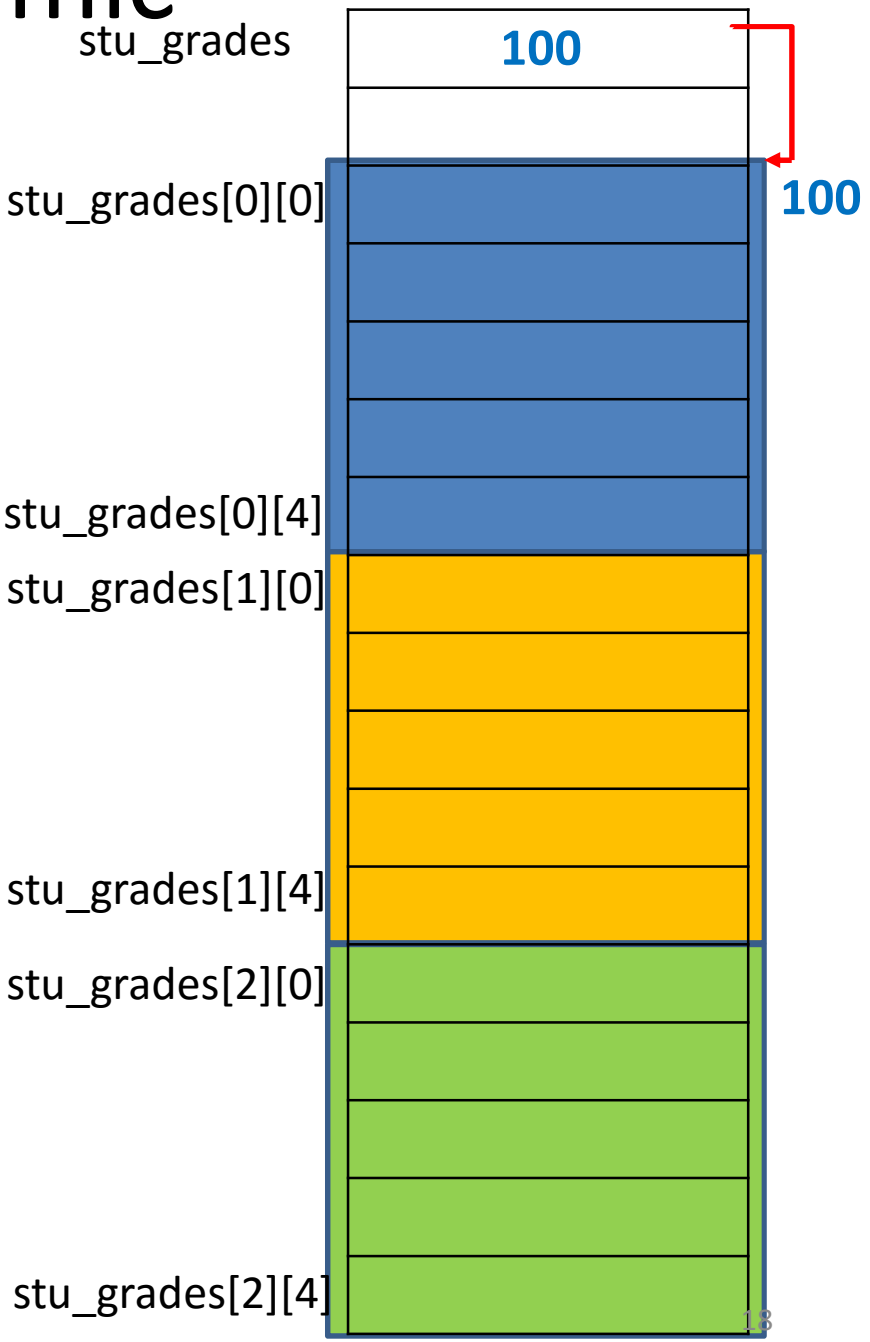
sizeof(***stu_grades**)

Allocated area is CONTIGUOUS

First dim dynamic

```
int main()
{  int (*stu_grades)[5];
   int no_of_stu, i, j, temp;
    scanf("%d", &no_of_stu);
    // assume 3
    stu_grades = (int (*)[5])
          malloc(sizeof(int [5])*3);


for (i=0; i< no_of_stu; i++)
      for (j=0; j<5; j++)
      {   scanf("%d", &temp);
          stu_grades[i][j] = temp; }}
```

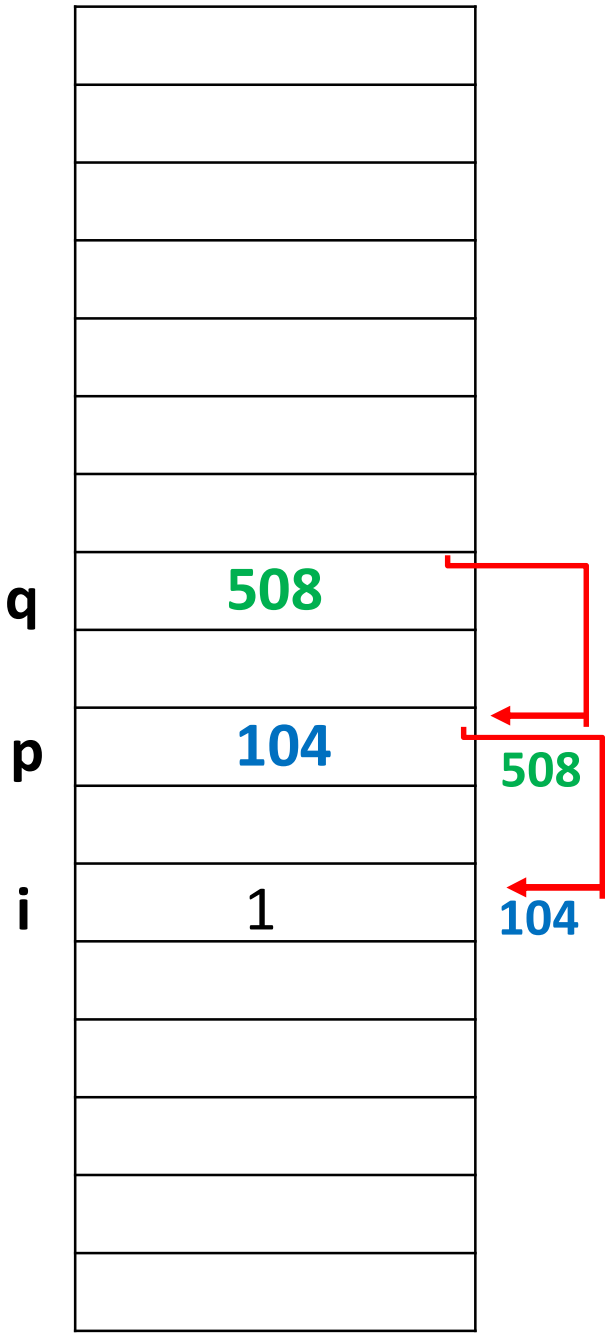


Case 1: First dim dynamic

```
int main()
{  int (*stu_grades)[5]; /* ptr to a block of 5 ints */
   …
   stu_grades = (int (*)[5]) malloc(sizeof(int [5])*no_of_stu);
   f(stu_grades, no_of_stu, 5);
}
```

Sending this array as a **parameter** to a function

```
  void f(int stu_grades[][5], int no_of_stu, int no_of_gra)
  void f(int (*stu_grades)[5], int no_of_stu, int no_of_gra)
  { stu_grades[i][j] = … }
```



Pointers to Pointers

int i =1;

int *p;

p = &i;

Assume some variable q

q=&p;

What is the type of q?

A ptr to ptr to integer!

printf("%d", i) OR

printf("%d", *p) OR

printf("%d", **q)

How to declare it?

int **q;

No limits on the levels of indirection!

Case 2: Second dim dynamic

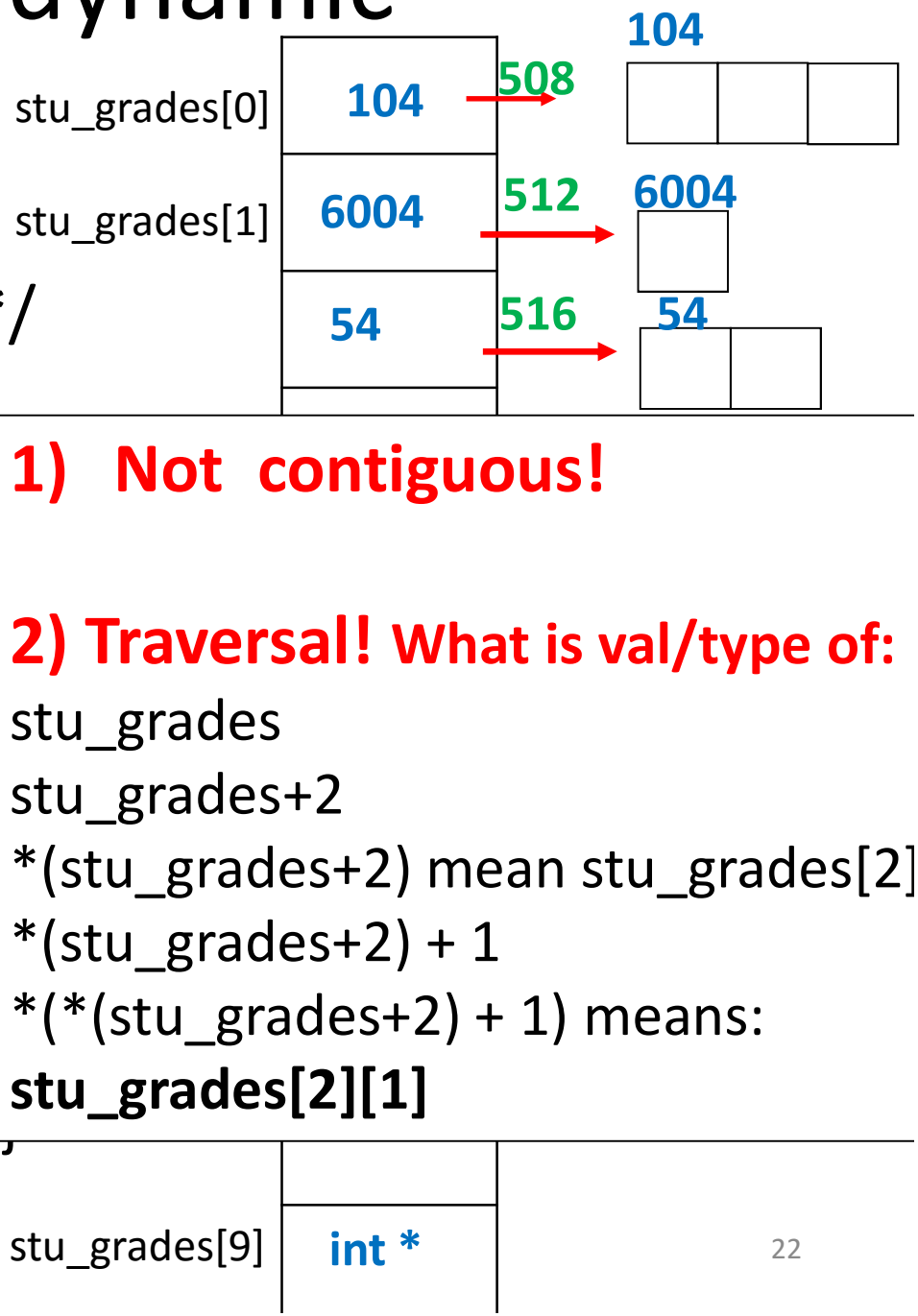
- I have 10 students, but number of grades per student wil be determined during run-time → iliffe (ragged) vector

Not contiguous!

```
int main()
{  int *stu_grades[10]; /* array of 10 pointers to int */
   int no_of_gra, i, j, temp;
   for (i=0; i< 10; i++)
   {  scanf("%d", & no_of_gra);
      stu_grades[i] = (int *) malloc(sizeof(int)*no_of_gra);
      for (j=0; j< no_of_gra; j++)
      {   scanf("%d", &temp);
          stu_grades[i][j] = temp; }
   }
}      /* Lets draw this on the board */
```

Case 2: Second dim dynamic

```
int main()
{  int *stu_grades[10];
   /* array of 10 pointers to int */
    int no_of_gra, i, j, temp;
    for (i=0; i< 10; i++)
    {  scanf("%d", & no_of_gra);
       stu_grades[i] = (int *)
malloc(sizeof(int)*no_of_gra);
       for (j=0; j< no_of_gra; j++)
       {   scanf("%d", &temp);
           stu_grades[i][j] = temp;
    } }
```



1) Not contiguous!

2) Traversal! What is val/type of:

stu_grades

stu_grades+2

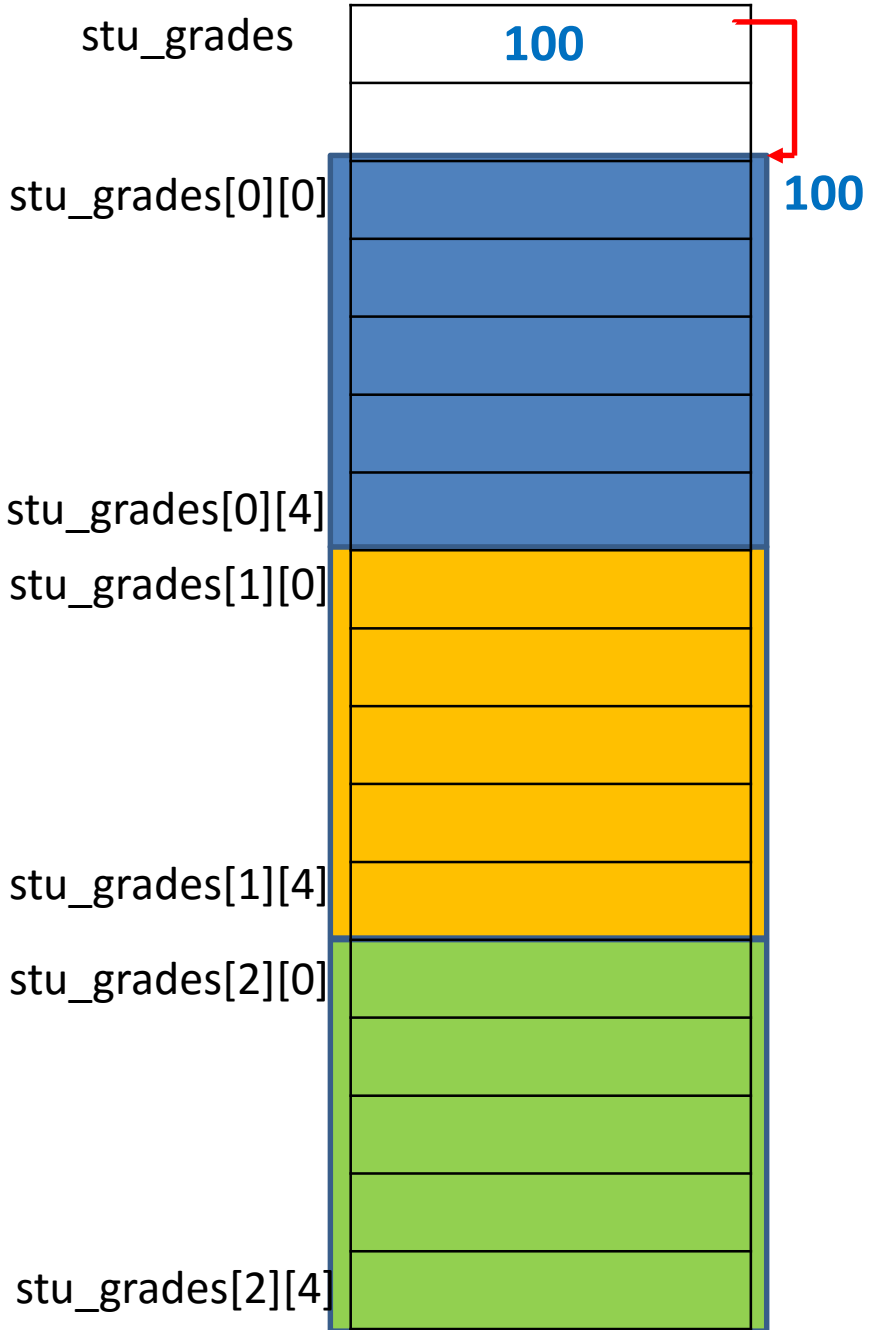
*(stu_grades+2) mean stu_grades[2]

*(stu_grades+2) + 1

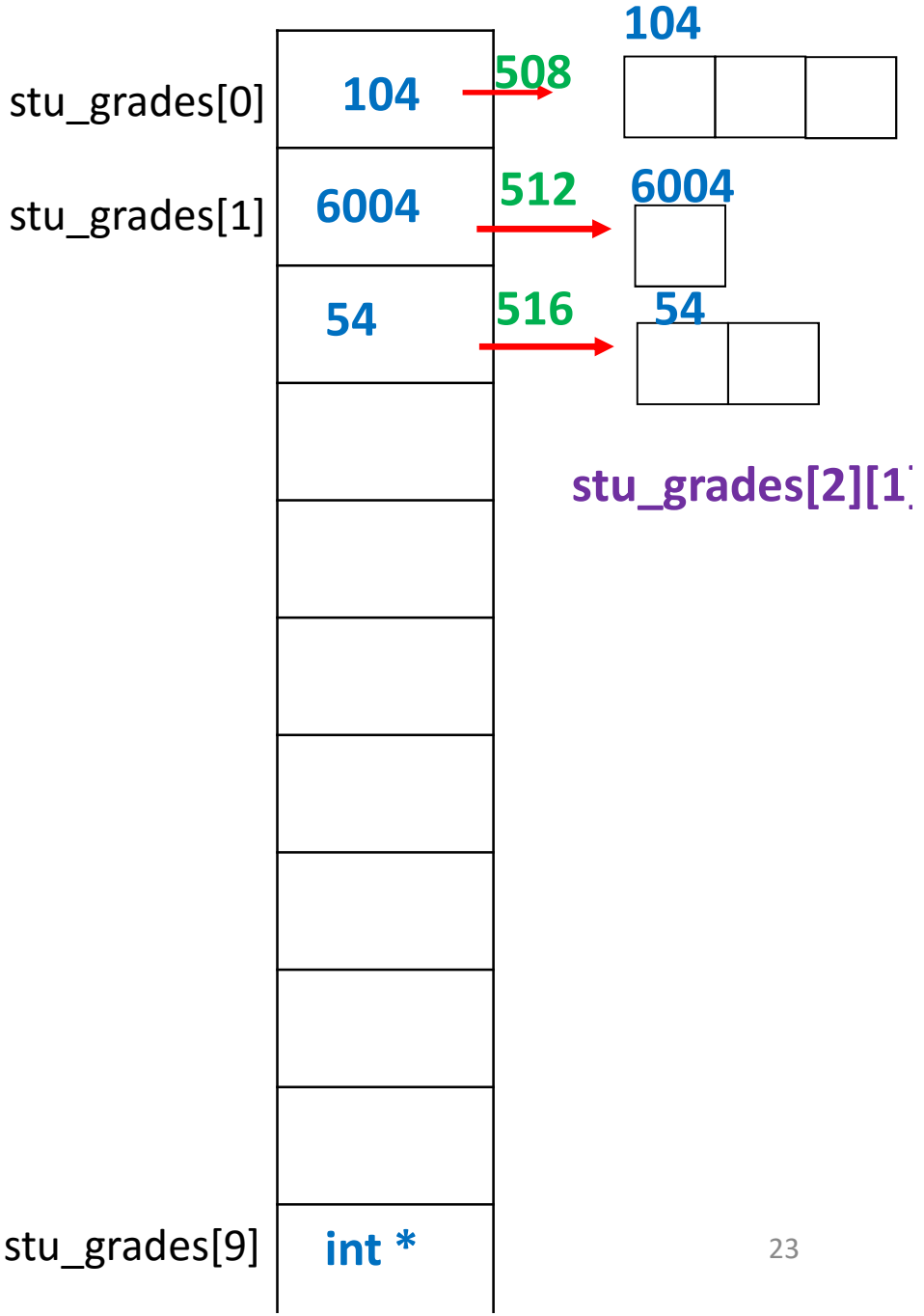
((stu_grades+2) + 1) means:

stu_grades[2][1]

int (*stu_grades)[5];



int *stu_grades[10];



Case 2: Second dim dynamic

```
int main()
{  int *stu_grades[10]; /* array of 10 pointers to int */
   …
   f(stu_grades, 10, …);
}
```

Sending this array as a **parameter** to a function

void f(**int *stu_grades[]**, …) → stu_grades is the ptr to first arrays element, which is of type int * so:

```
void f(int **stu_grades, …)
{ stu_grades[i][j] = … }
```

In main or f(), notation is the same while accessing array elements, but what really happens is slightly different than Case1 or true-2D array

Case 3: Both dim lenghts dynamic

- Both number of students and grades per student wil be determined during run-time

Not contiguous!

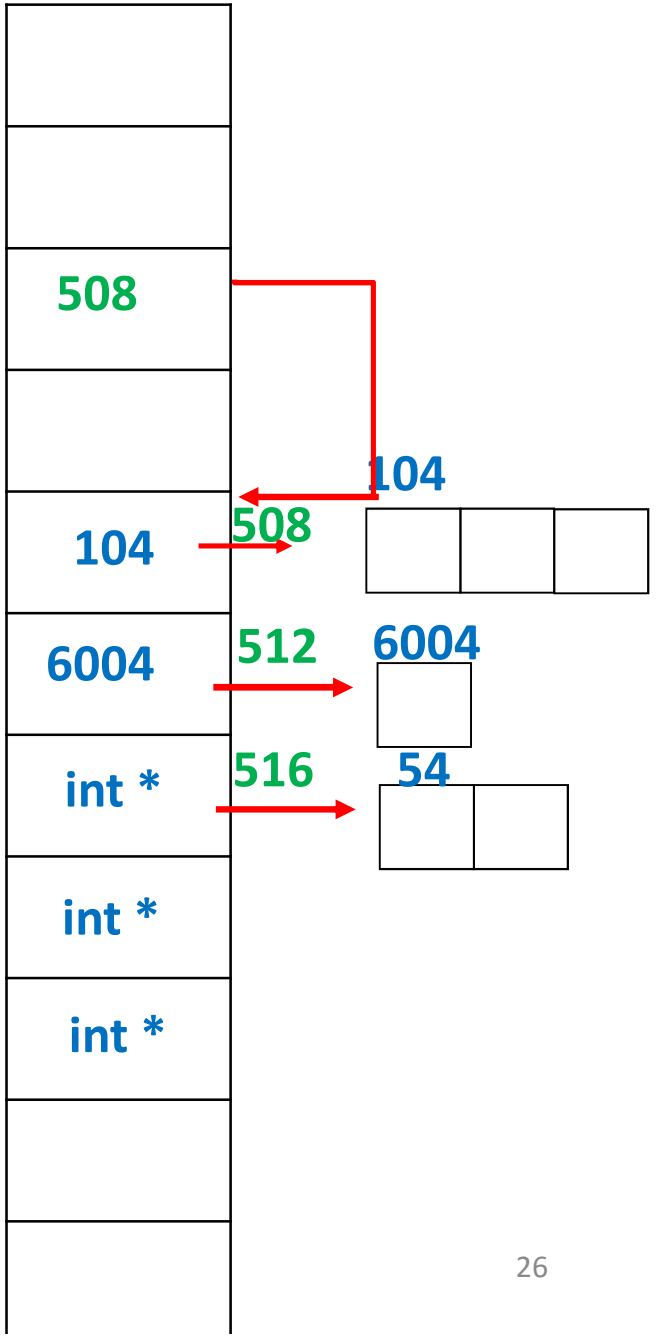
```
int main()
{  int **stu_grades;
   int no_of_gra, no_of_stu, i, j, temp;
   scanf("%d", &no_of_stu);
   stu_grades = (int **) malloc(sizeof(int *)*no_of_stu);
   for (i=0; i< no_of_stu; i++)
   {  scanf("%d", & no_of_gra);
      stu_grades[i] = (int *) malloc(sizeof(int)*no_of_gra);
      for (j=0; j< no_of_gra; j++)
      {   scanf("%d", &temp);  stu_grades[i][j] = temp; }
   } }      /* Lets draw this on the board */
```

Case 3: Both dynamic

```
int main()
{  int **stu_grades;
   int no_of_gra, no_of_stu, i, j, temp;

   scanf("%d", &no_of_stu);
   stu_grades = (int **) malloc(sizeof(int *)*no_of_stu);

   for (i=0; i< no_of_stu; i++)
   {  scanf("%d", & no_of_gra);
      stu_grades[i] = (int *) malloc(sizeof(int)*no_of_gra);
      for (j=0; j< no_of_gra; j++)
      {   scanf("%d", &temp);  stu_grades[i][j] = temp; }
    }
}
```



Case 3: Both dim lenghts dynamic

```
int main()
{  int **stu_grades;
    …
   f(stu_grades, …, …);
}
```

Sending this array as a **parameter** to a function

```
   void f(int **stu_grades, …)
   { stu_grades[i][j] = … }
```

Recall: In Case 2 and Case 3 we should also store the no of grades of grades per student to be able to access them correctly later…