Dynamic Memory Allocation

Basic Idea

- In some situations, data is dynamic in nature.
  - Amount of data cannot be predicted beforehand.
  - Number of data item keeps changing during program execution.
- Dynamic Memory Allocation: Ability of a program to use more memory space at execution time
  - Memory space required can be specified at the time of execution.
  - C supports allocating and freeing memory dynamically using library routines.

Dynamic Memory Allocation

- Ability of a program to use more memory space at execution time
  - Hold new nodes
    - Use function **`malloc`** to allocate memory
  - Release space no longer needed
    - Use function **`free`** to deallocate memory
  - Use **`#include <stdlib.h>`** header file when using malloc & free functions

Memory Usage + Heap

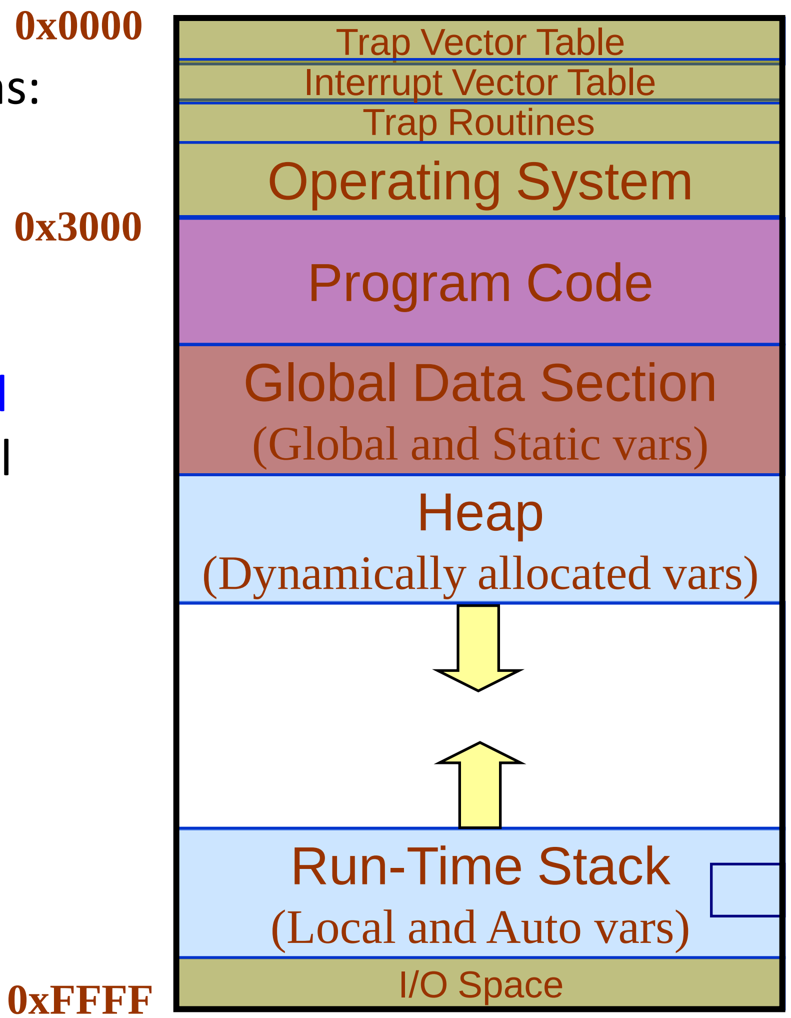
Variable memory is allocated in three areas:

- Global data section
- Run-time stack
- Dynamically allocated - heap

Global variables are allocated in the global data section and are accessible from all parts of the program.

Local variables are allocated during execution on the run-time stack.

Dynamically allocated variables are items created during run-time and are allocated on the heap.



Memory Allocation Functions

- malloc
  - Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
- calloc
  - Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
- free

  Frees previously allocated space.
- realloc
  - Modifies the size of previously allocated space.

Functions `malloc` & `free`

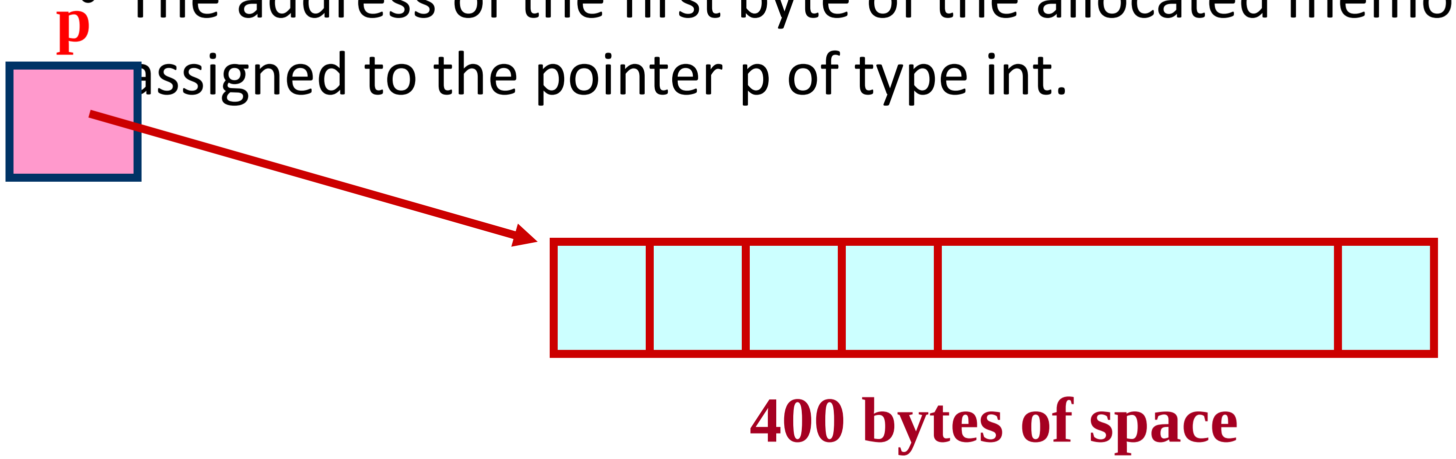
- **ptr = malloc(sizeof(struct node));**
  - sizeof(struct node) returns the size in bytes of the structure
  - malloc() allocates the specified number of bytes in memory
  - Returns a pointer to the place in memory
  - Returns NULL, if no more memory is available
- **free(ptr);**
  - Deallocates the memory referred to by the pointer so it can be reused

Contd.

- Examples

  p = (int *) malloc (100 * sizeof (int)) ;

  - A memory space equivalent to "100 times the size of an int" bytes is reserved.
  - The address of the first byte of the allocated memory is assigned to the pointer p of type int.



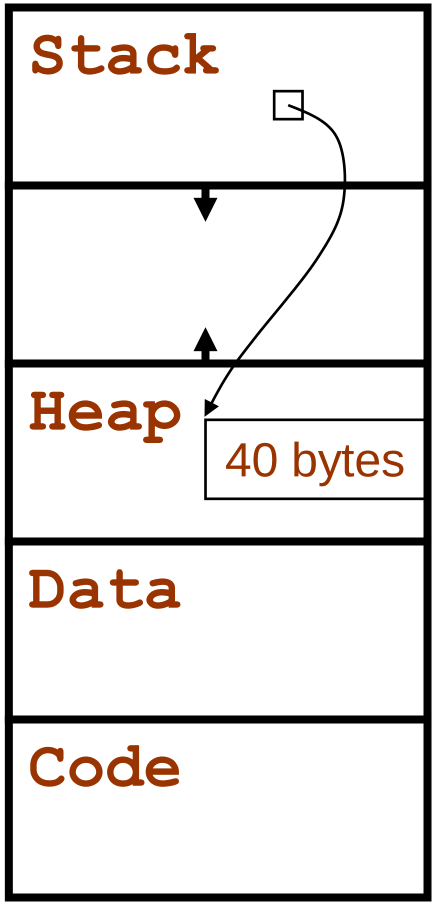
malloc

```
int *ip;
ip = (int *) malloc(10 * sizeof(int));
If (ip == NULL)
{
    /* Handle Error! */
}
```

- Options for handling error
  - Abort
  - Ask again
  - Save user data
  - Ask for less
  - Free up something

malloc -- What happens?

```
int foo(int n) {
  int *ip;
  ip = malloc(n * sizeof(int));
  if(ip == NULL) {
    /* Handle Error! */
  }
  ...
```



```
if((ip = malloc(10*sizeof(int))) == NULL)
{
   /* Handle Error Here */
}
```

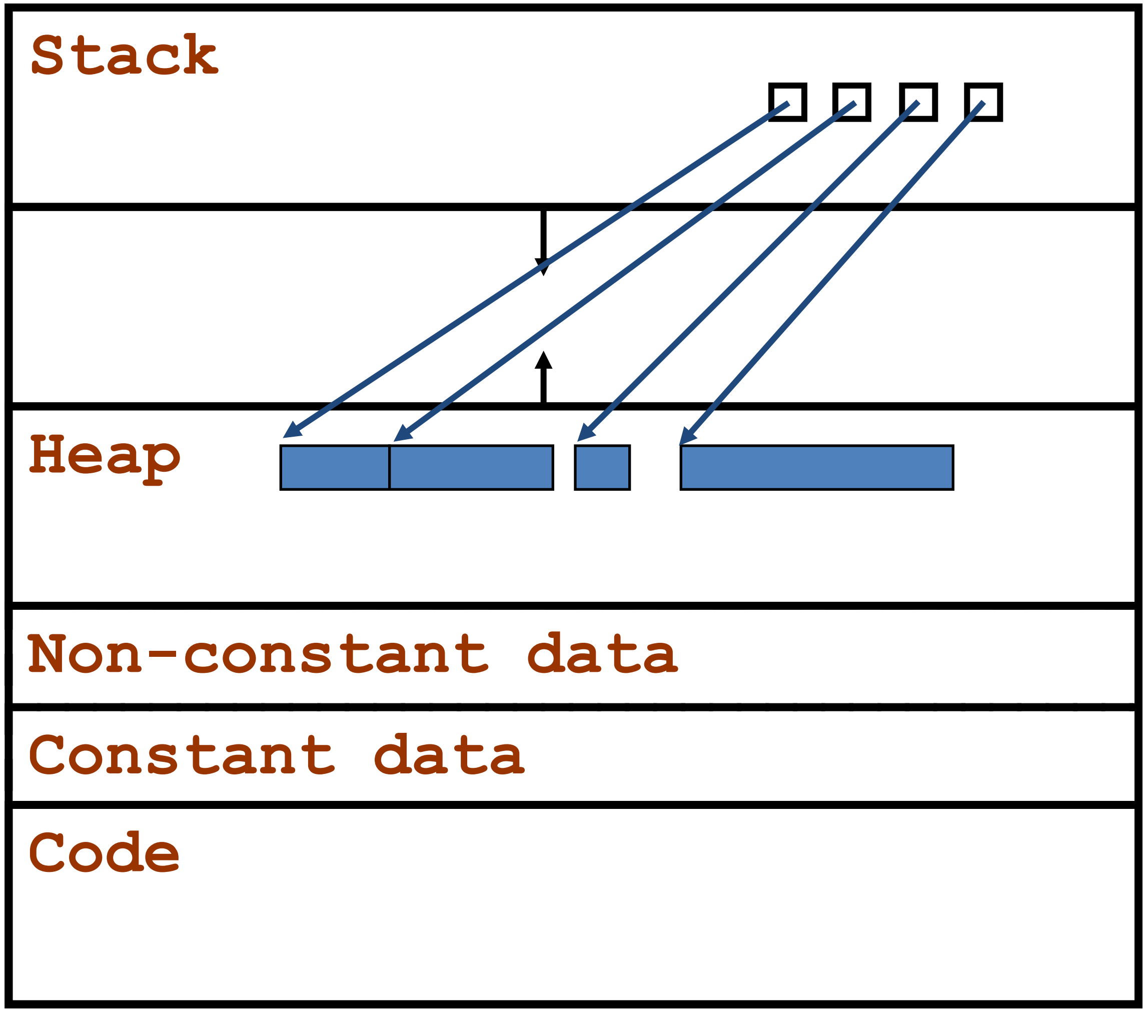
Using the space

```
int sample(int n)
{
    int i;
    int *ip;
    if((ip = malloc(n*sizeof(int))) == NULL)
    {
        /* Handle Error Here */
    }
    for(i = 0; i < n; i++)
        ip[i] = 0;
...
```

Flexibility

```
#define MAX 10
int *ip;
ip = malloc(MAX * sizeof(int));
```

After some calls to malloc



What does runtime track?

Address	Size
1400	200
2400	300
3000	100
4000	500

Notice that no record is made of the name of any pointer

What happens?

```
int *ip;
ip = malloc(...);
    .
    .
    .
free(ip);
```

Prototypes

```
void *malloc(size_t n);

void free(void *p);

void *realloc(void *p, size_t n);
```

- What is this mysterious void pointer?

void pointer

- Not originally in c
- Relatively recent addition
- Basically a “generic” pointer

- Intended for use in applications like free where the block of memory located at some address will be freed without any necessity of defining the type

Using `calloc()` (1)

- calloc()is a variant of malloc()
- calloc() takes two arguments: the number of "things" to be allocated and the size of each "thing" (in bytes)
- calloc() returns the address of the chunk of memory that was allocated
- calloc() also sets all the values in the allocated memory to zeros (malloc() doesn't)

Using `calloc()` (2)

- `calloc()` is also used to dynamically allocate arrays
- For instance, to dynamically allocate an array of 10 `ints`:

```
int *arr;
arr = (int *) calloc(10, sizeof(int));
/* now arr has the address
   of an array of 10 ints, all 0s */
```

`malloc/calloc` return value (1)

- **`malloc`** and **`calloc`** both return the address of the newly-allocated block of memory
- However, they are *not* guaranteed to succeed!
  - maybe there is no more memory available
- If they fail, they return **`NULL`**
- You must *always* check for NULL when using **`malloc`** or **`calloc`**

`malloc/calloc` return value (2)

- bad:

```
int *arr = (int *) malloc(10 * sizeof(int));
/* code that uses arr... */
```

- good:

```
int *arr = (int *) malloc(10 * sizeof(int));
if (arr == NULL) {
    fprintf(stderr, "out of memory!\n");
    exit(1);
}
```

- Always do this!

`malloc()` vs. `calloc()`

- **`malloc/calloc`** both allocate memory
- **`calloc()`** zeros out allocated memory, **`malloc()`** doesn't.

Functions `malloc` & `free`: Examples

- Once allocated, the memory belongs to your program until it terminates or is `free()`'d.

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

int main() {
	int* dynArray;

	/* Allocate space for 16 ints */
	dynArray = (int *)malloc( 16 * sizeof(int) );
	dynArray[6] = 65;
	dynArray[12] = 2;
	doSomething( dynArray );
	free( dynArray );
}
```

Using `free()` (2)

```
int *arr;
arr = (int *) calloc(10, sizeof(int));

/* now arr has the address  of an array of 10 ints, all 0s */

/* Code that uses the array... */

/* Now we no longer need the array, so "free" it: */
free(arr);
/* Now we can't use arr anymore. */
```

Using `free()` (3)

- NOTE: When we **`free()`** some memory, the memory is *not* erased or destroyed
- Instead, the operating system is informed that we don't need the memory any more, so it may use it for *e.g.* another program
- Trying to use memory after freeing it can cause a segmentation violation (program crash)

Dynamic memory allocation (3)

```
#include <stdlib.h>
int *foo(int n) {
  int i[10];  /* memory allocated here */
  int i2[n];  /* ERROR: NOT VALID! */
  int *j;
  j = (int *)malloc(n * sizeof(int));
  /* Alternatively: */
  /* j = (int *)calloc(n, sizeof(int)); */
  return j;
} /* i’s memory deallocated here; j’s not */
```

Dynamic memory allocation (4)

```
void bar(void) {
  int *arr = foo(10);
  arr[0] = 10;
  arr[1] = 20;
  /* ... do something with arr ... */
  free(arr);  /* deallocate memory */
}
```

- Not calling `free()` leads to *memory leaks* !

Structures and `malloc`

```
struct person {
	char initials[4];
	long int ssn;
	int height;
	struct person *father;
	struct person *mother;
} *tom, *bill, *susan;

int main() {
	tom = (struct person *)malloc( sizeof( struct person ) );
	bill = (struct person *)malloc( sizeof( struct person ) );
	susan = (struct person *)malloc( sizeof( struct person ) );

	strncpy(tom->initials, "tj“, 2);
	tom->ssn = 555235512;
	tom->father = bill;
	tom->mother = susan;
	susan->height = 68;
	/* Since tom is now a pointer, tom->mother->height is correct. */
	printf(“\nTom's mother's height is: %d", tom->mother->height);
}
```

Structures and `malloc`

```
struct person {
    char initials[4];
    long int ssn;
    int height;
    struct person *father;
    struct person *mother;
} *tom, *bill, *susan;
```

Structures and `malloc`

```
int main( ) {
    tom = (struct person *)malloc( sizeof( struct person ) );
    bill = (struct person *)malloc( sizeof( struct person ) );
    susan = (struct person *)malloc( sizeof( struct person ) );

    strncpy(tom->initials, "tj“, 2);
    tom->ssn = 555235512;
    tom->father = bill;
    tom->mother = susan;
    susan->height = 68;

    printf(“\nTom's mother's height: %d", tom->mother->height);
}
```

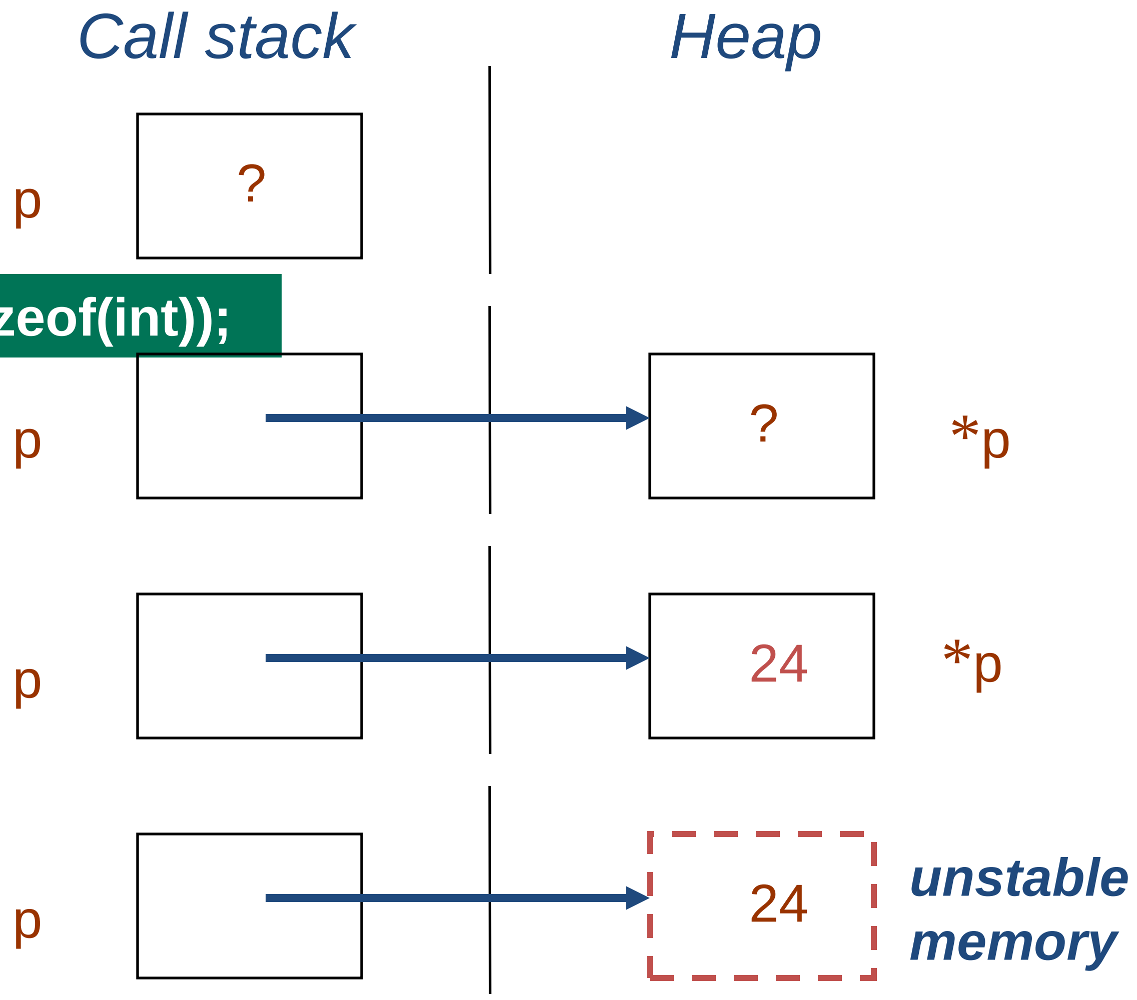
Dynamic Allocation – Example 1

```
int *p;
```

```
p = (int*)malloc(sizeof(int));
```

```
*p = 24;
```

```
free(p);
```



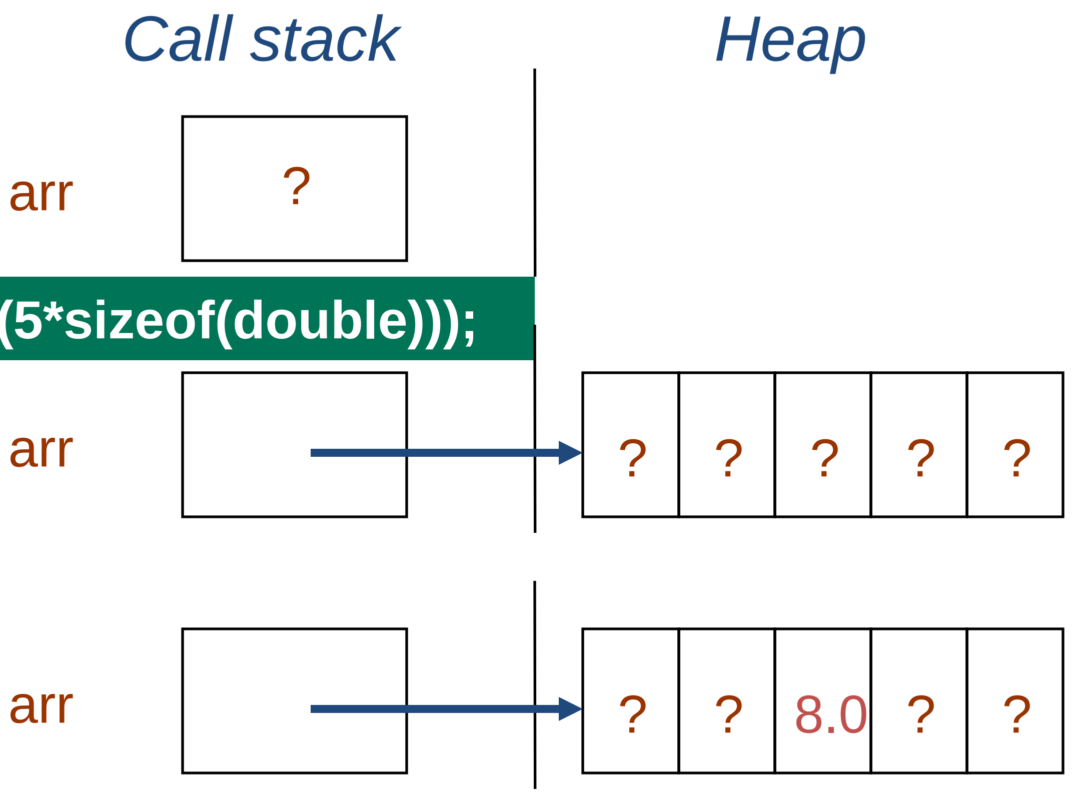
Dynamic Allocation – Example 2

Pointer to an array:

```
double * arr;
```

```
arr = (double*)(malloc(5*sizeof(double)));
```

```
arr[2] = 8.0;
```

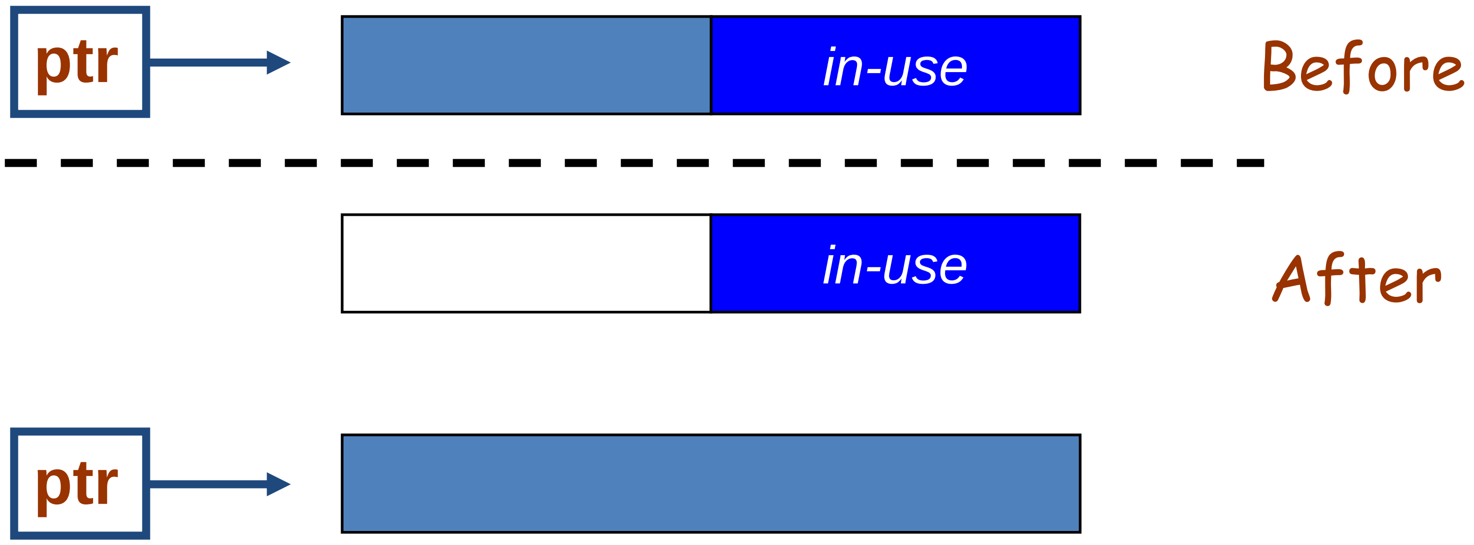


Realloc

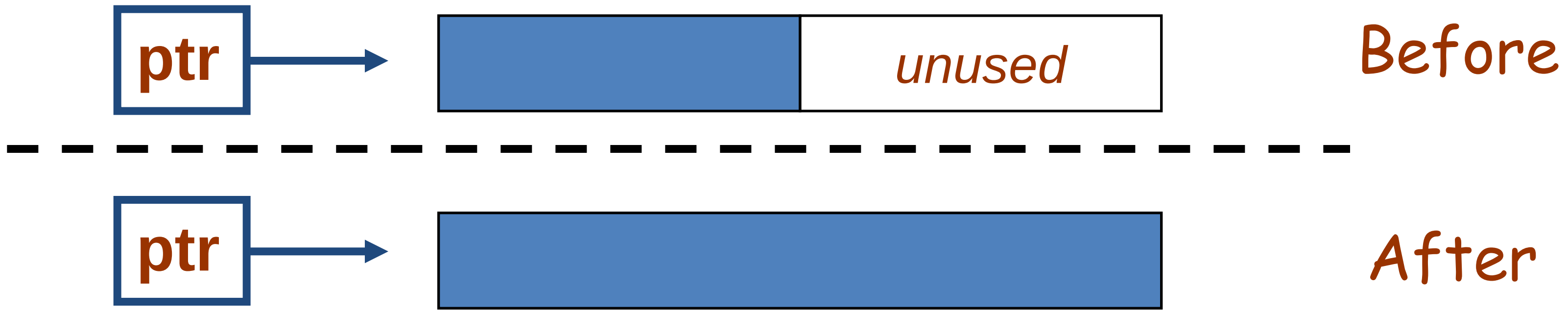
```
ptr = realloc(ptr, num_bytes);
```

- What it does (conceptually)
  - Find space for new allocation
  - Copy original data into new space
  - Free old space
  - Return pointer to new space

Realloc



Realloc: What might happen



```
void read_array (int *a, int n) ;
int sum_array (int *a, int n) ;
void wrt_array (int *a, int n) ;

int  main ()  {
        int *a, n;
        printf (“Input n: “) ;
        scanf (“%d”, &n) ;
        a = calloc (n, sizeof (int)) ;
        read_array (a, n) ;
        wrt_array (a, n) ;
        printf (“Sum = %d\n”, sum_array(a, n);
}
```

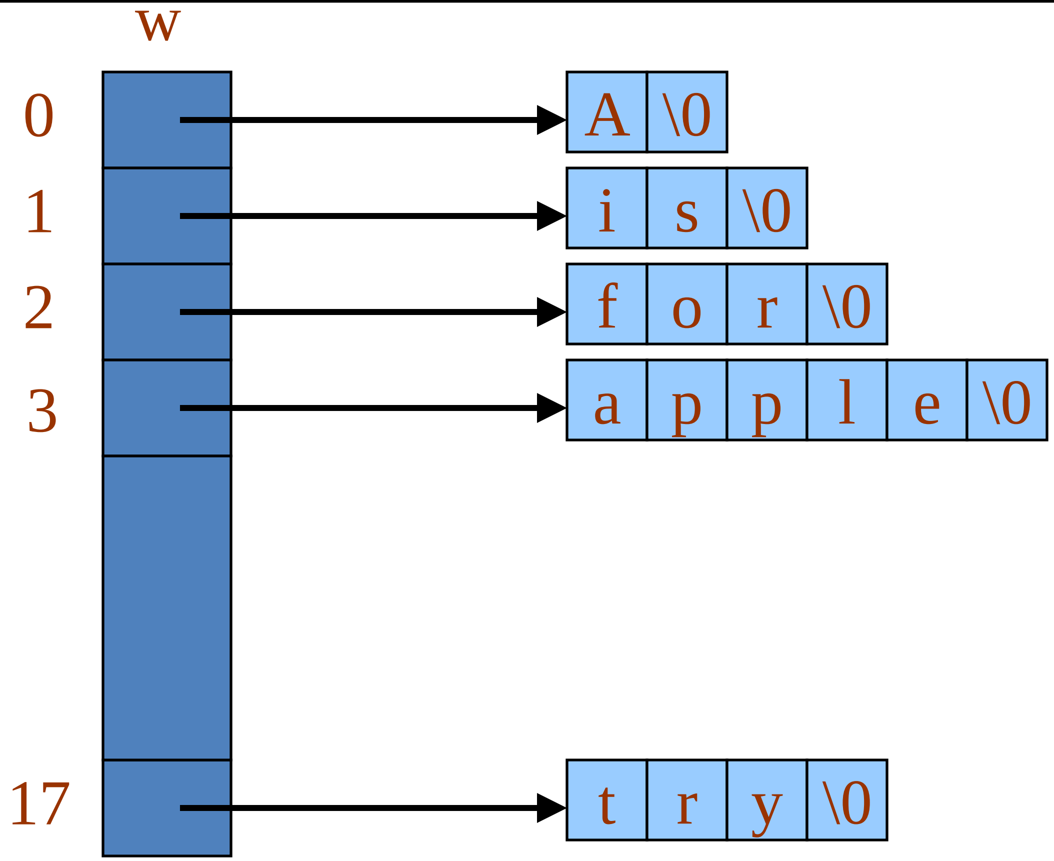
```
void read_array (int *a, int n) {
        int i;
        for (i=0; i<n; i++)
                scanf (“%d”, &a[i]) ;
}
void sum_array (int *a, int n) {
         int i, sum=0;
         for (i=0; i<n; i++)
                sum += a[i] ;
         return sum;
}
void wrt_array (int *a, int n) {
         int i;
         ........
}
```

Arrays of Pointers

- Array elements can be of any type
  - array of structures
  - array of pointers

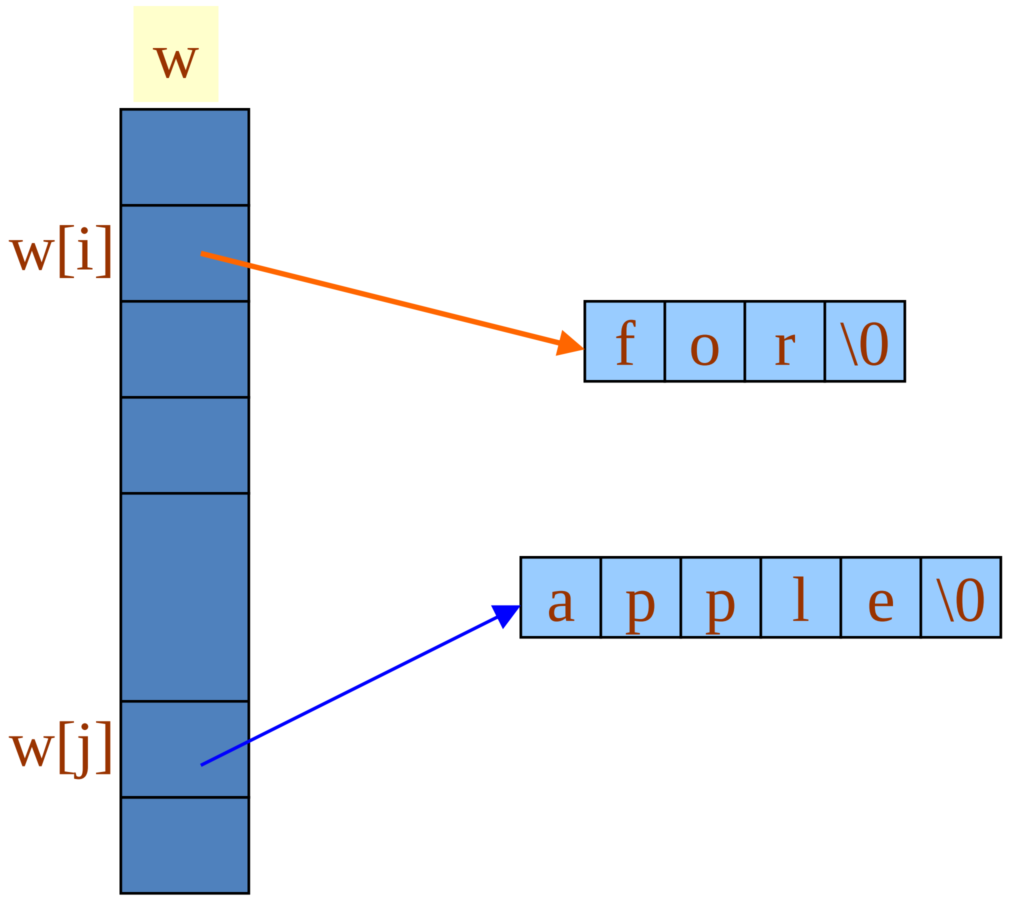
```
int main (void)  {
        char word[MAXWORD];
        char * w[N];      /* an array of pointers */
        int i, n;               /* n: no of words to sort */
        for (i=0; scanf("%s", word) == 1); ++i) {
            w[i] = calloc (strlen(word)+1, sizeof(char));
            if (w[i] == NULL) exit(0);
              strcpy (w[i], word) ;
        }
        n = i;
        sortwords (w, n) ;
        wrt_words (w, n);
        return 0;
}
```

Input : A is for apple or alphabet pie which
all get a slice of come taste it and try

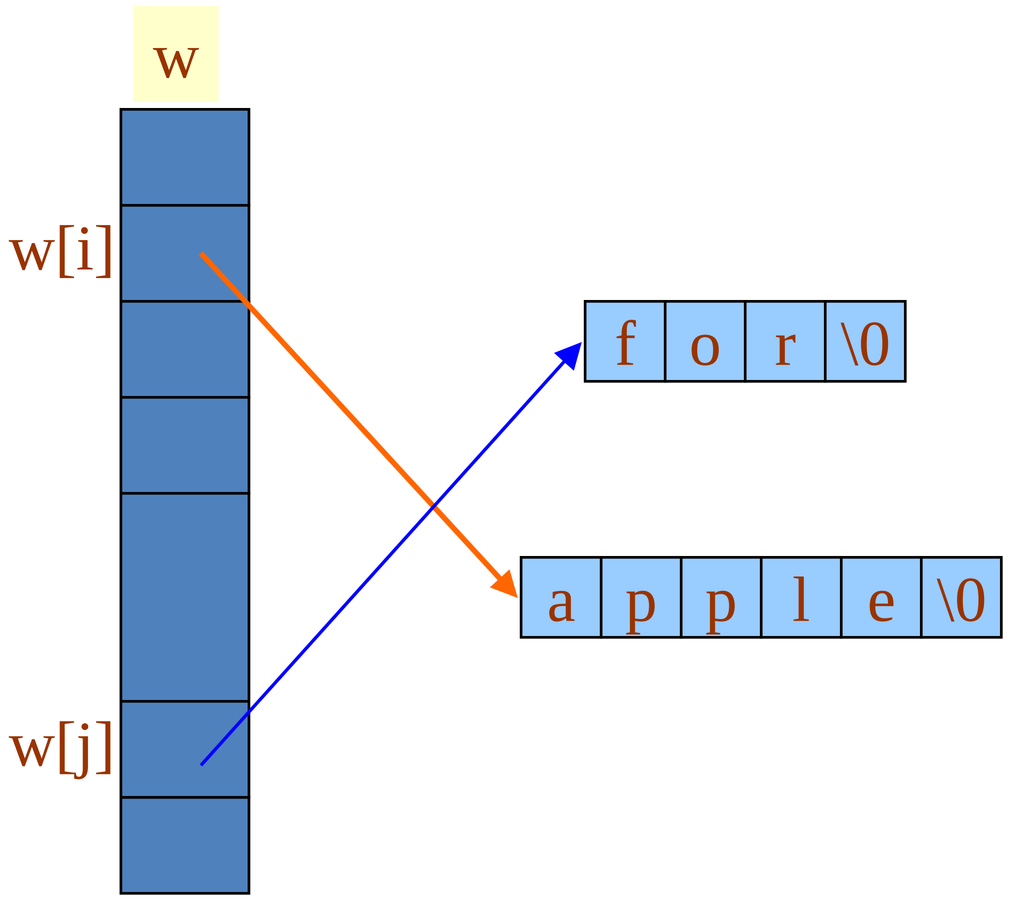


```
void sort_words (char *w[], int n) {
        int i, j;
        for (i=0; i<n; ++i)
              for (j=i+1; j<n; ++j)
                     if (strcmp(w[i], w[j]) > 0)
                           swap (&w[i], &w[j]) ;
}
void swap (char **p, char **q)  {
        char *tmp ;
        tmp = *p;
        *p = *q;
        *q = tmp;
}
```

Before swapping



After swapping



Example

```
#include <stdio.h>

int main() {
  int i,N;
  float *height;
  float sum=0,avg

  printf("Input th
  scanf("%d",&N);

  height=(float *) malloc(N * sizeof(float));
```

```
Input the number of students.
5
Input heights for 5 students
23 24 25 26 27
Average height= 25.000000
```

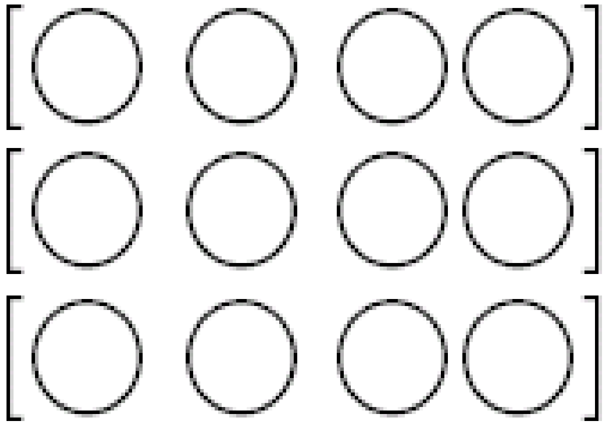
```
printf("Input heights for %d
udents \n",N);
or(i=0;i<N;i++)
scanf("%f",&height[i]);

or(i=0;i<N;i++)
  sum+=height[i];

 avg=sum/(float) N;

 printf("Average height= %f \n",
avg);
}
```

2d arrays and pointers

- Recall that when we declare a two-dimensional array, we can think of it as an array of arrays. For example, if we have the following array declaration,

  int data[3][4];

- we can think of this as an array with three members, each of which is an array of four ints. We can visualize it like this:



- The address of the beginning of the entire array is the same as the address of the first row of the array, which is the address of the first element of the first row.
- However, to get to the first row we must dereference the array name and to get the value of the first element of the first row we must dereference twice

```
int sales[2][3] = { {1, 2, 3},
                    {9, 10, 11} };
printf("address of data is %p\n", sales);
printf("address of row 0 of data is %p\n", *sales);
printf("the value of sales[0][0] is %d\n", **sales);
```

produces

```
address of data is          0x7fffffed8140
address of row 0 of data is 0x7fffffed8140
the value of sales[0][0] is             1
```

2d arrays

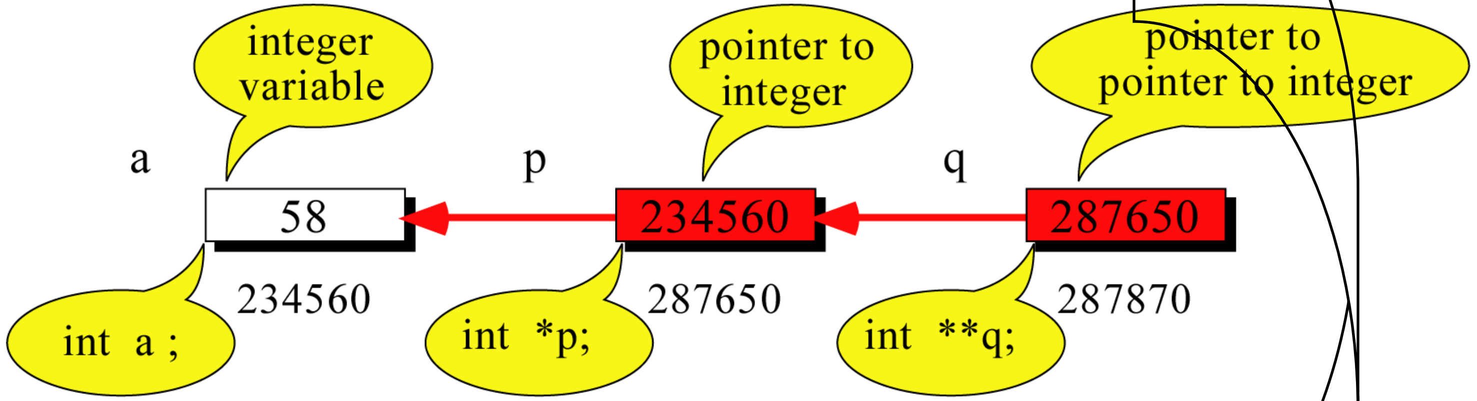
- The general form for getting the address of any element of any row is

  ***(array_name + row) + column**

- For example, when we write *(data + 1) + 2, we are saying “add the size of one row to the address of data, get the address of this, then add the size of two elements of a row to this”.

Pointer Indirection *(Pointers to Pointers)*

```
//  Local Declarations
int        a ;
int       *p ;
int      **q ;
```



```
a=58;
p=&a;
q=&p;
```

a = 58

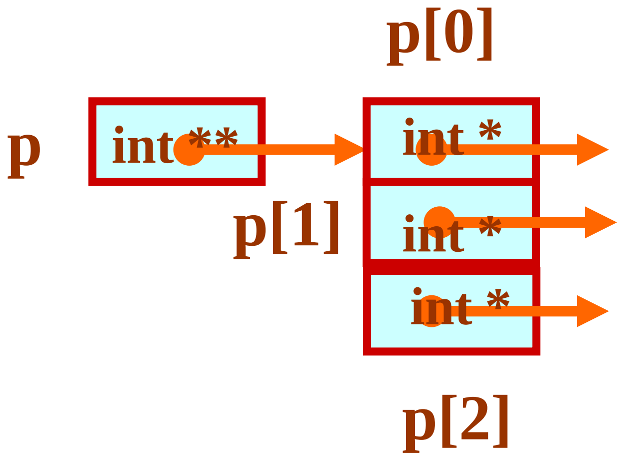
*p = 58

**q = 58

Pointer to Pointer

- Example:

```
int **p;

p=(int **) malloc(3 * sizeof(int *));
```



2d array

int AA[MAX][15]

- A two dimensional array is considered to be a one dimensional array of rows, which are, themselves, one dimensional arrays.
- The rows are stored in sequence, starting with row 0.

AA[0] is stored first, then AA[1], then AA[2], and so on to AA[MAX-1].

- Each of these ``elements'' is an array.
- The same is true for higher dimensional arrays.
- A n-dimensional array is considered to be a one dimensional array whose elements are, themselves, arrays of dimension.

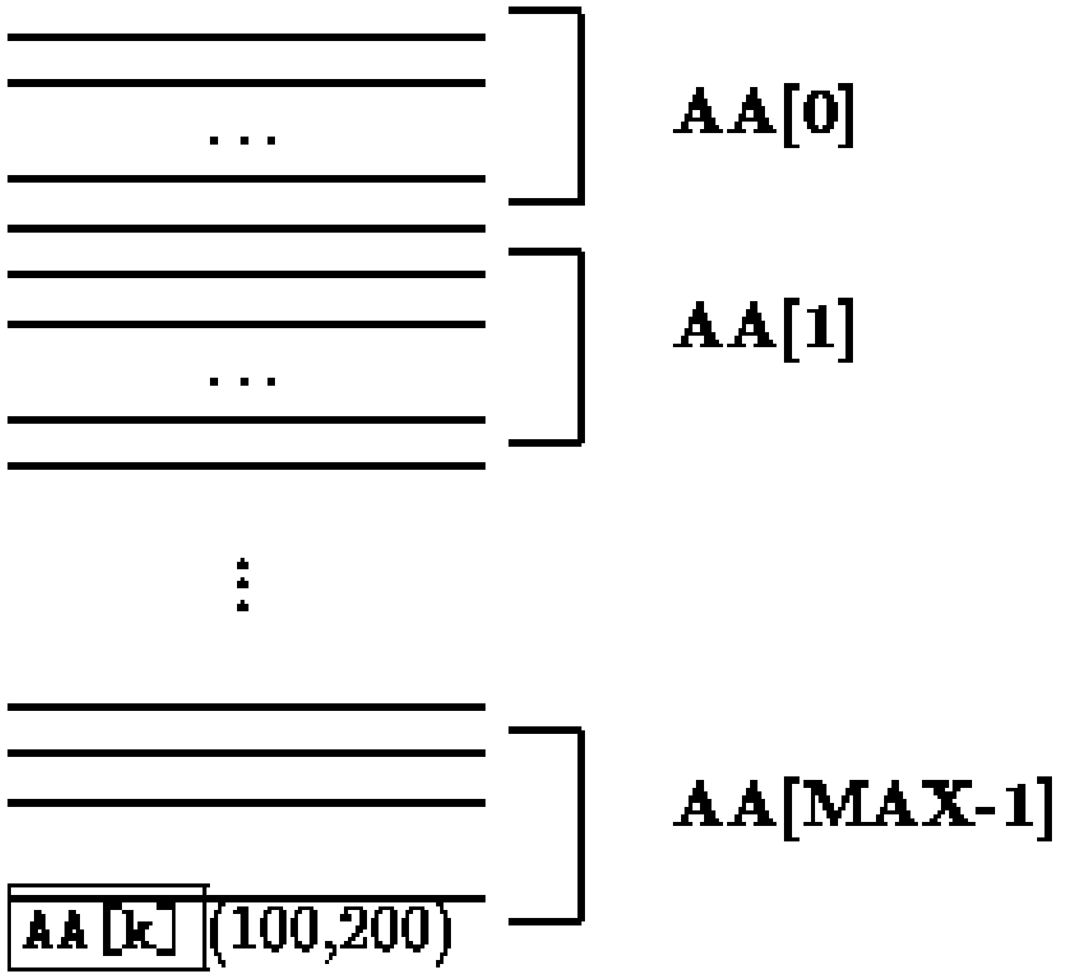


Figure 9.10: A Two Dimensional Array in Row Major Order

2d array

- Recall that an array name (without an index) represents a pointer to the first object of the array.
- So the name, AA, is a pointer to the element AA[0].
- But, AA[0] is a one dimensional array; so, AA[0] points to the first object in row 0, i.e. AA[0] points to AA[0][0].
- AA[k] is the address of AA[k][0].
- **AA[k] + j** points to **AA[k][j]**, and ***(AA[k] + j)** accesses the value of **AA[k][j]** .

2d array

- The name, AA points to the first object in this array of arrays, i.e. AA points to the array AA[0].
- The addresses represented by AA and AA[0] are the same; however, they point to objects of different types.
- AA[0] points to AA[0][0], so it is an integer pointer.
- AA points to AA[0], so it is a pointer to an integer pointer.
- If we add 1 to AA, the resulting pointer, AA + 1, points to the array AA[1], and AA + k points to the array AA[k].
- Adding 1 to AA[0] results in a pointer that points to AA[0][1]

Pointer equivalence to 2d arrays

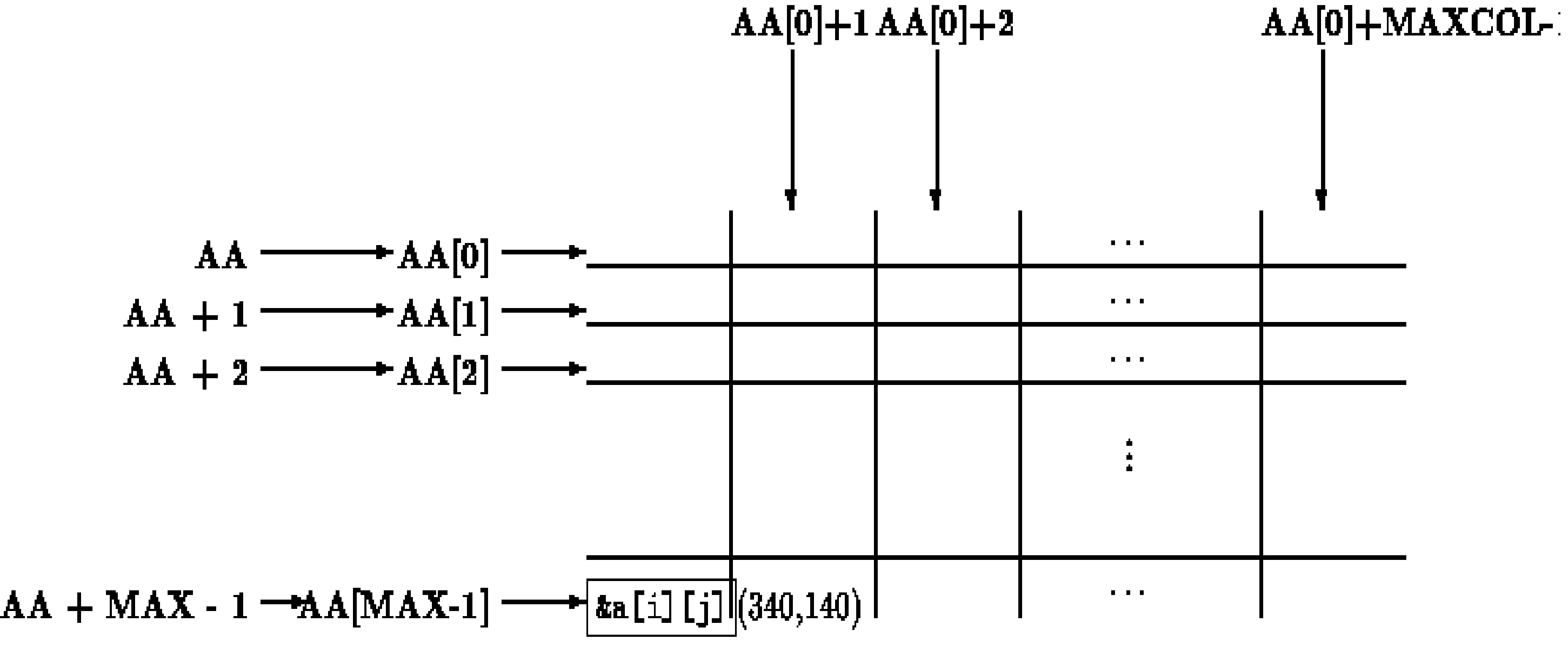


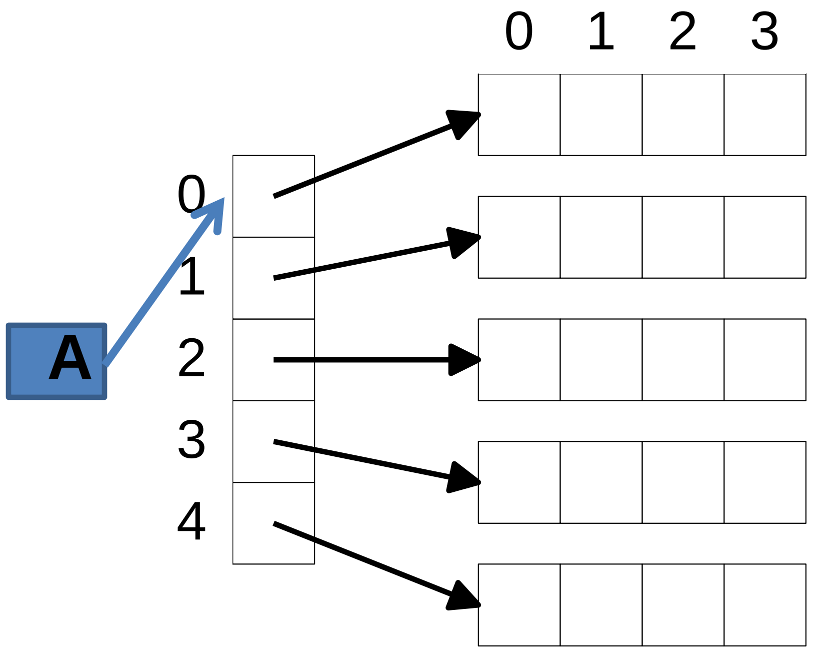
Figure 9.11: Pointers and Two Dimensional Arrays

Dynamically Allocating 2D Arrays

Can not simply dynamically allocate 2D (or higher) array

Idea - allocate an array of pointers (first dimension), make each pointer point to a 1D array of the appropriate size

Can treat result as 2D array



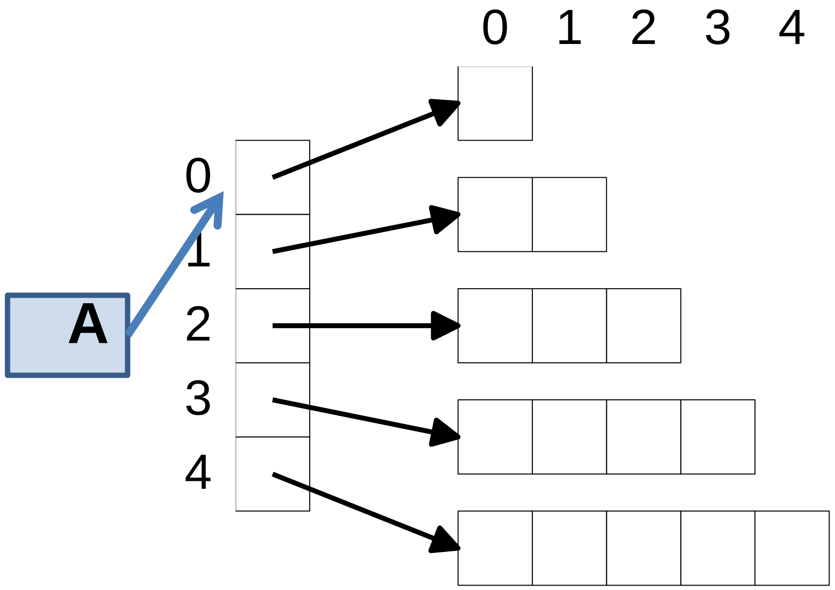
Dynamically Allocating 2D Array

```
float **A;  /* A is an array (pointer) of float
               pointers */
int I;

A = (float **) calloc(5,sizeof(float *));
/* A is a 1D array (size 5) of float pointers */

for (I = 0; I < 5; I++)
  A[I] = (float *) calloc(4,sizeof(float));
/* Each element of array points to an array of 4
   float variables */

/* A[I][J] is the Jth entry in the array that the
   Ith member of A points to */
```

Non-Square 2D Arrays

No need to allocate square 2D arrays:

```
float **A;
int I;

A = (float **) calloc(5,
              sizeof(float *));

for (I = 0; I < 5; I++)
  A[I] = (float **)
               calloc(I+1,
                sizeof(float));
```



Dynamically Allocating Multidimensional Arrays

2-D Array Allocation

```
#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

```
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]);
 }
```

Elements accessed like 2-D array elements.

2-D Array: Contd.

```
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");
    }
}
```

```
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
```

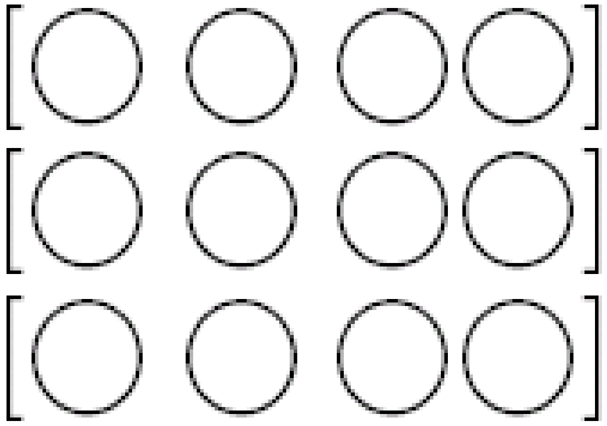
```
int main()
{
  int **p;
  int M,N;

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

2d arrays and pointers

- Recall that when we declare a two-dimensional array, we can think of it as an array of arrays. For example, if we have the following array declaration,

```
int data[3][4];
```

- we can think of this as an array with three members, each of which is an array of four ints. We can visualize it like this:



2d arrays

- The general form for getting the address of any element of any row is

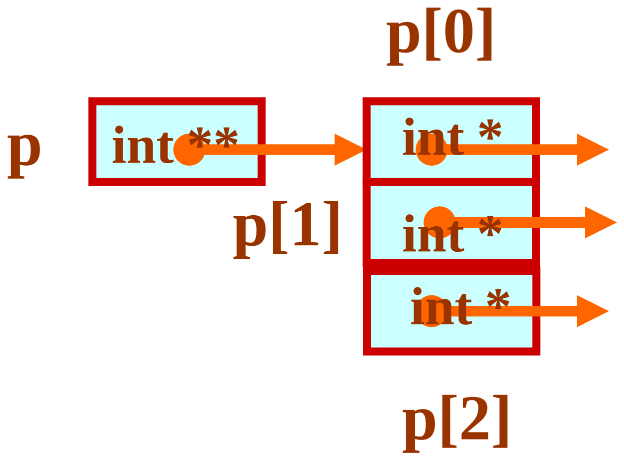
  ***(array_name + row) + column**

- For example, when we write *(data + 1) + 2, we are saying “add the size of one row to the address of data, get the address of this, then add the size of two elements of a row to this”.

Pointer to Pointer

- Example:

```
int **p;
p=(int **) malloc(3 * sizeof(int *));
```



2d array

int AA[MAX][15]

- A two dimensional array is considered to be a one dimensional array of rows, which are, themselves, one dimensional arrays.
- The rows are stored in sequence, starting with row 0.

AA[0] is stored first, then AA[1], then AA[2], and so on to AA[MAX-1].

- Each of these ``elements'' is an array.
- The same is true for higher dimensional arrays.
- A n-dimensional array is considered to be a one dimensional array whose elements are, themselves, arrays of dimension.

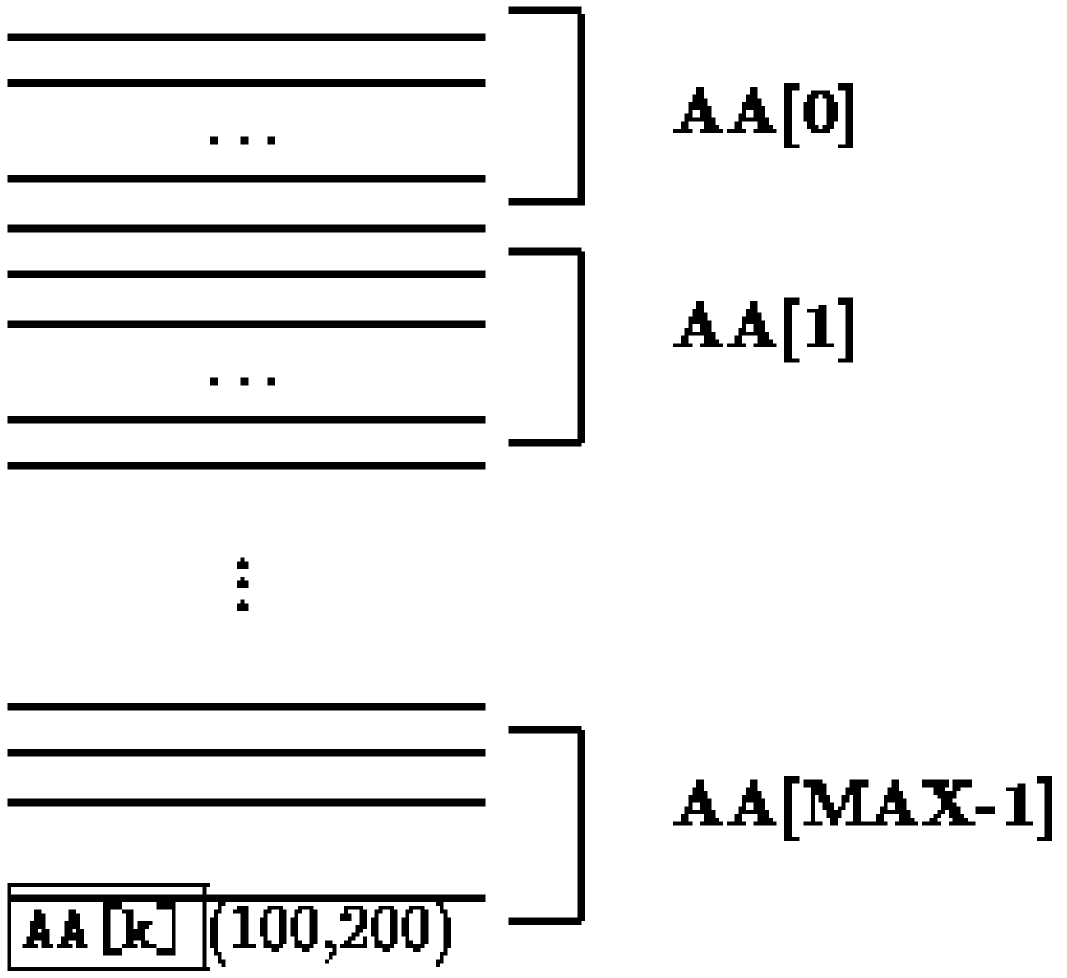


Figure 9.10: A Two Dimensional Array in Row Major Order

2d array

- Recall that an array name (without an index) represents a pointer to the first object of the array.
- So the name, AA, is a pointer to the element AA[0].
- But, AA[0] is a one dimensional array; so, AA[0] points to the first object in row 0, i.e. AA[0] points to AA[0][0].
- AA[k] is the address of AA[k][0].
- **AA[k] + j** points to **AA[k][j]**, and ***(AA[k] + j)** accesses the value of **AA[k][j]** .

2d array

- The name, AA points to the first object in this array of arrays, i.e. AA points to the array AA[0].
- The addresses represented by AA and AA[0] are the same; however, they point to objects of different types.
- AA[0] points to AA[0][0], so it is an integer pointer.
- AA points to AA[0], so it is a pointer to an integer pointer.
- If we add 1 to AA, the resulting pointer, AA + 1, points to the array AA[1], and AA + k points to the array AA[k].
- Adding 1 to AA[0] results in a pointer that points to AA[0][1]

Pointer equivalence to 2d arrays

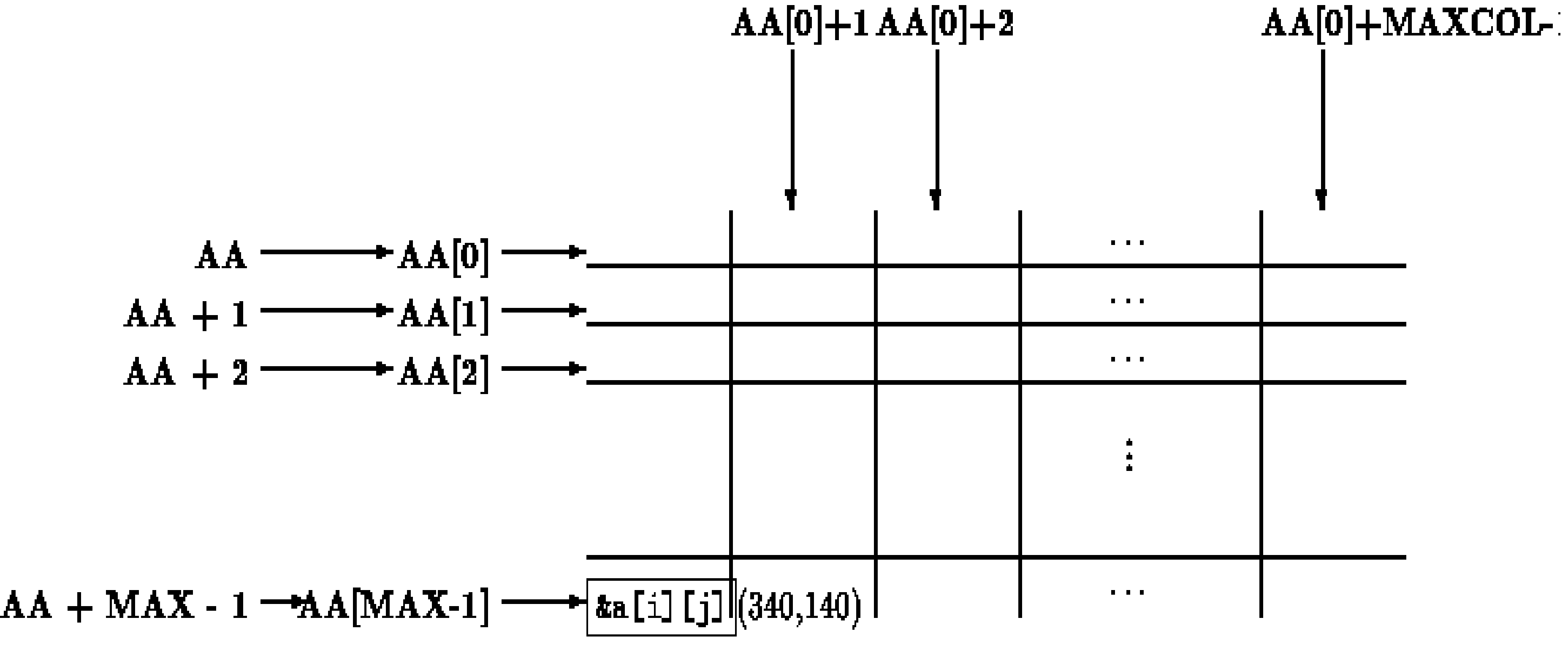


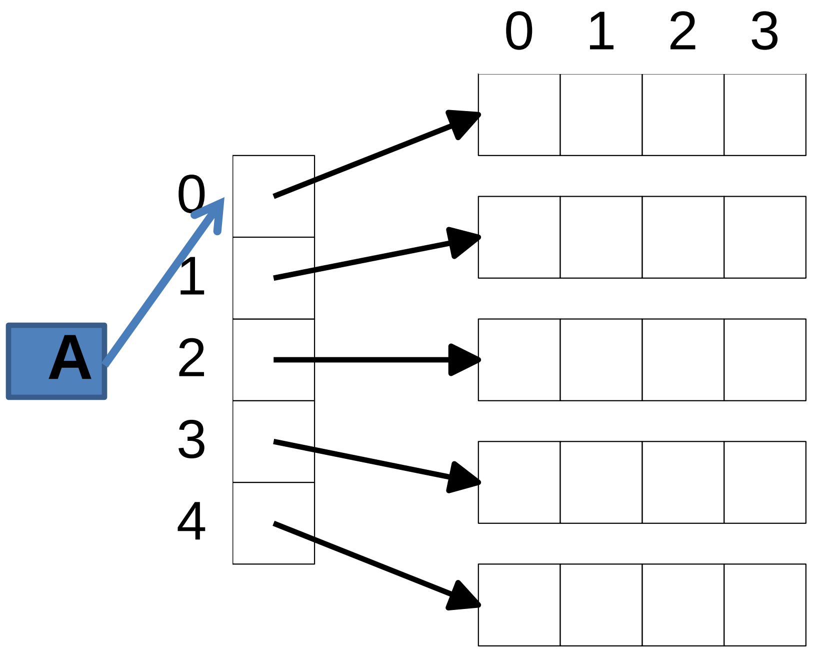
Figure 9.11: Pointers and Two Dimensional Arrays

Dynamically Allocating 2D Arrays

Can not simply dynamically allocate 2D (or higher) array

Idea - allocate an array of pointers (first dimension), make each pointer point to a 1D array of the appropriate size

Can treat result as 2D array



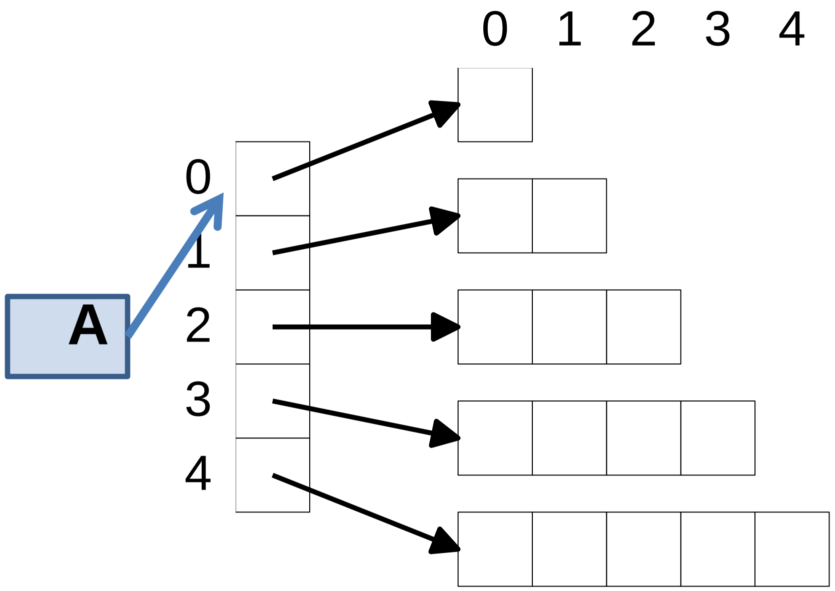
Dynamically Allocating 2D Array

```
float **A;  /* A is an array (pointer) of float
               pointers */
int I;

A = (float **) calloc(5,sizeof(float *));
/* A is a 1D array (size 5) of float pointers */

for (I = 0; I < 5; I++)
  A[I] = (float *) calloc(4,sizeof(float));
/* Each element of array points to an array of 4
   float variables */

/* A[I][J] is the Jth entry in the array that the
   Ith member of A points to */
```

Non-Square 2D Arrays

No need to allocate square 2D arrays:

```
float **A;
int I;

A = (float **) calloc(5,
             sizeof(float *));

for (I = 0; I < 5; I++)
  A[I] = (float **)
              calloc(I+1,
               sizeof(float));
```



2-D Array Allocation

```
#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

```
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]);
  }
```

Elements accessed like 2-D array elements.

2-D Array: Contd.

```
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");
     }
}
```

```
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
```

```
int 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("\n The array read as \n");
  print_data(p,M,N);
```

Array Pointers

- To declare a pointer to an array type, you must use parentheses

```
int (* arrPtr)[10] ;     // A pointer to an array of
                         // ten elements with type int
```

int *a[10];

- Declares and allocates an array of pointers to int. Each element must be dereferenced individually.

int (*a)[10];

- Declares (without allocating) a pointer to an array of int(s). The pointer to the array must be dereferenced to access the value of each element.
- **int a[10];** Declares and allocates an array of int(s).

Array Pointers

```
int (* arrPtr)[10] ;     // A pointer to an array of 10 elements
                         // with type int
```

if we assign it the address of an appropriate array,

- *arrPtr yields the array, and
- (*arrPtr)[i] yields the array element with the index i.

```
int matrix[3][10]; // Array of 3 rows, each with 10 columns.
                   // The array name is a pointer to the first row.
arrPtr = matrix;   // Let arrPtr point to the first row of the matrix.
(*arrPtr)[0] = 5; // Assign the value 5 to the first element of the first row.
arrPtr[2][9] = 6; // Assign 6 to the last element of the last row.
++arrPtr; // Advance the pointer to the next row.
(*arrPtr)[0] = 7; // Assign 7 to the first element of the second row.
++arrPtr; // Advance the pointer to the next row.
(*arrPtr)[0] = 7; // Assign value 7 to the first element of the second row.
```

```
int main ( ) {
    int i;
    int * a[10] ;
    printf (“sizeof a = %d\n” , sizeof(a));
    int x=1, y=2, z=3;
    a[0] = &x; a[1] = &y; a[2] = &z;
    for (i=0; i<10; i++)
            printf ("*a[%d] = %d\n", i, *(a[i])) ;
}
```

```
sizeof a = 80
*a[0] = 1
*a[1] = 2
*a[2] = 3
```

```
int main ( ) {
    int i;
    int (*b)[10] ;
    printf (“sizeof b = %d\n”, sizeof(b))
    b = malloc (10*sizeof(int)) ;
    printf ("b = %p\n", b) ;
    printf ("b+1 = %p\n", b+1) ;
    for (i=0; i<10; i++) {
        (*b)[i] = i;
        printf (“(*b)[%d] = %d\n",i, (*b)[i]) ;
    }
}
```

```
sizeof  b = 8
b = 0x601010
b+1 = 0x601038
(*b)[0] = 0
(*b)[1] = 1
(*b)[2] = 2
(*b)[3] = 3
(*b)[4] = 4
```