Dynamic Memory Allocation

Allocate memory on demand...

Dynamic Memory Allocation

Problem with arrays

Sometimes

- Amount of data cannot be predicted beforehand
- Number of data items keeps changing during program execution

Example: Search for an element in an array of N elements

One solution: find the maximum possible value of N and allocate an array of N elements

- Wasteful of memory space, as N may be much smaller in some executions
- Example: maximum value of N may be 10,000, but a particular run may need to search only among 100 elements
  - Using array of size 10,000 always wastes memory in most cases

Better solution

Dynamic memory allocation

- Know how much memory is needed after the program is run
  - Example: ask the user to enter from keyboard
- Dynamically allocate only the amount of memory needed

C provides functions to dynamically allocate memory

- malloc, calloc, realloc

Dynamic Memory Allocation

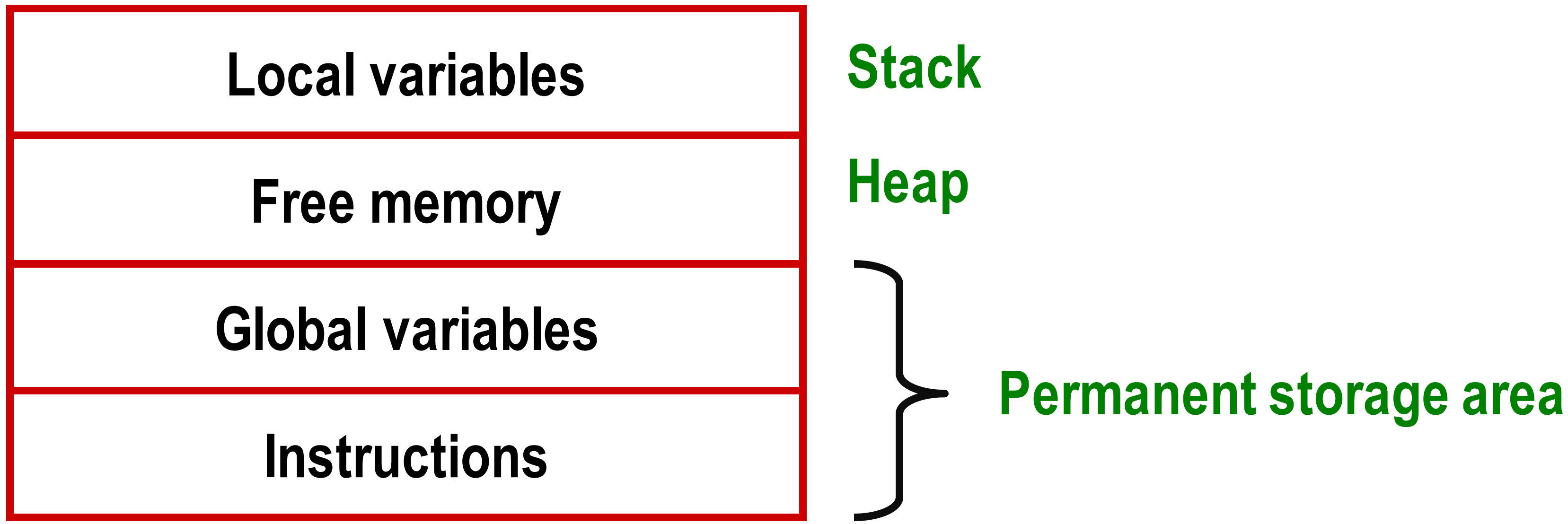
Normally the number of elements in an array is specified in the program

- **Often leads to wastage or memory space or program failure.**

Dynamic Memory Allocation

- **Memory space required can be specified at the time of execution.**
- **C supports allocating and freeing memory dynamically using library routines.**

Memory Allocation Process in C



Memory Allocation Process

The program instructions and the global variables are stored in a region known as *permanent storage area*.

The local variables are stored in another area called *stack*.

The memory space between these two areas is available for dynamic allocation during execution of the program.

- This free region is called the *heap*.
- The size of the heap keeps changing.

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.

Allocating a Block of Memory

A block of memory can be allocated using the function malloc.

- Reserves a block of memory of specified size and returns a pointer of type void.
- The return pointer can be type-casted to any pointer type.

General format:

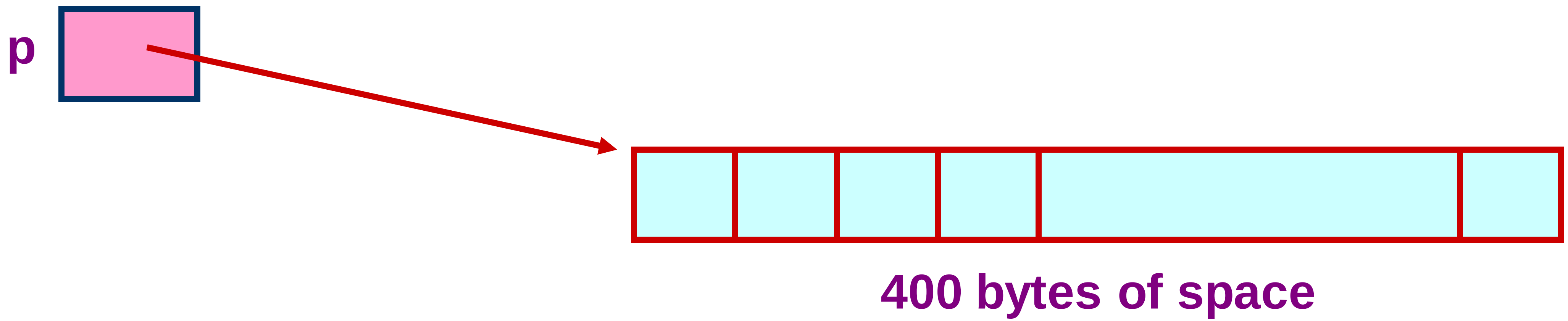
```
ptr =  (type *) malloc (byte_size);
```

Allocating a Block of Memory

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`.



Allocating a Block of Memory

```
cptr = (char *) malloc (20);
```

- Allocates 20 bytes of space for the pointer cptr of type char.

```
sptr = (struct stud *) malloc (10 * sizeof (struct stud));
```

- Allocates space for a structure array of 10 elements. sptr points to a structure element of type “struct stud”.

Points to Note

`malloc` always allocates a block of contiguous bytes.

- The allocation can fail if sufficient contiguous memory space is not available.
- If it fails, `malloc` returns NULL.

```
if  ((p = (int *) malloc(100 * sizeof(int))) == NULL)
{
        printf ("\n Memory cannot be allocated");
        exit( ) ;
}
```

Can we allocate only arrays?

malloc can be used to allocate memory for single variables also

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

- Allocates space for a single int, which can be accessed as *p
  - Single variable allocations are just special case of array allocations
    - Array with only one element

malloc( )-ing array of structures

```
typedef  struct{
        char name[20];
        int roll;
        float SGPA[8], CGPA;
    } person;
int main()  {
    person *student;
    int i,j,n;
    scanf("%d", &n);
    student = (person *)malloc(n*sizeof(person));
    for (i=0; i<n; i++) {
       scanf("%s", student[i].name);
       scanf("%d", &student[i].roll);
       for(j=0;j<8;j++) scanf("%f", &student[i].SGPA[j]);
       scanf("%f", &student[i].CGPA);
    }
    return 0;
}
```

Altering the Size of a Block

Sometimes we need to alter the size of some previously allocated memory block.

- More memory needed.
- Memory allocated is larger than necessary.

How?

- By using the `realloc` function.

If the original allocation is done as:

```
ptr = malloc (size);
```

then reallocation of space may be done as:

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

Altering the Size of a Block

- The new memory block may or may not begin at the same place as the old one.
    - If it does not find space, it will create it in an entirely different region and move the contents of the old block into the new block.
- The function guarantees that the old data remains intact.
- If it is unable to allocate, it returns `NULL` and frees the original block.

Using the malloc'd Array

Once the memory is allocated, it can be used with pointers, or with array notation

Example:

```
        int *p, n, i;
        scanf("%d", &n);
        p = (int *) malloc (n * sizeof(int));
    for (i=0; i<n; ++i)
            scanf("%d", &p[i]);
```

The n integers allocated can be accessed as *p, *(p+1), *(p+2),…, *(p+n-1) or just as p[0], p[1], p[2], …,p[n-1]

Example

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

  printf("Input no. of students\n");
  scanf("%d", &N);

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

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

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

  avg = sum / (float) N;

  printf("Average height = %f \n",
                avg);
  free (height);
  return 0;
}
```

Releasing the allocated space: free

An allocated block can be returned to the system for future use by using the free function

General syntax:

```
free (ptr);
```

where ptr is a pointer to a memory block which has been previously created using malloc

Note that no size needs to be mentioned for the allocated block, the system remembers it for each pointer returned

Arrays of Pointers

Static array of pointers

```
#define  N   20
#define  M  10
int main()
{
    char word[N], *w[M];
    int i, n;
    scanf("%d",&n);
    for (i=0; i<n; ++i) {
        scanf("%s",  word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
        strcpy (w[i], word) ;
    }
    for (i=0; i<n; i++) printf("w[%d]  = %s  \n",i,w[i]);
    return 0;
}
```

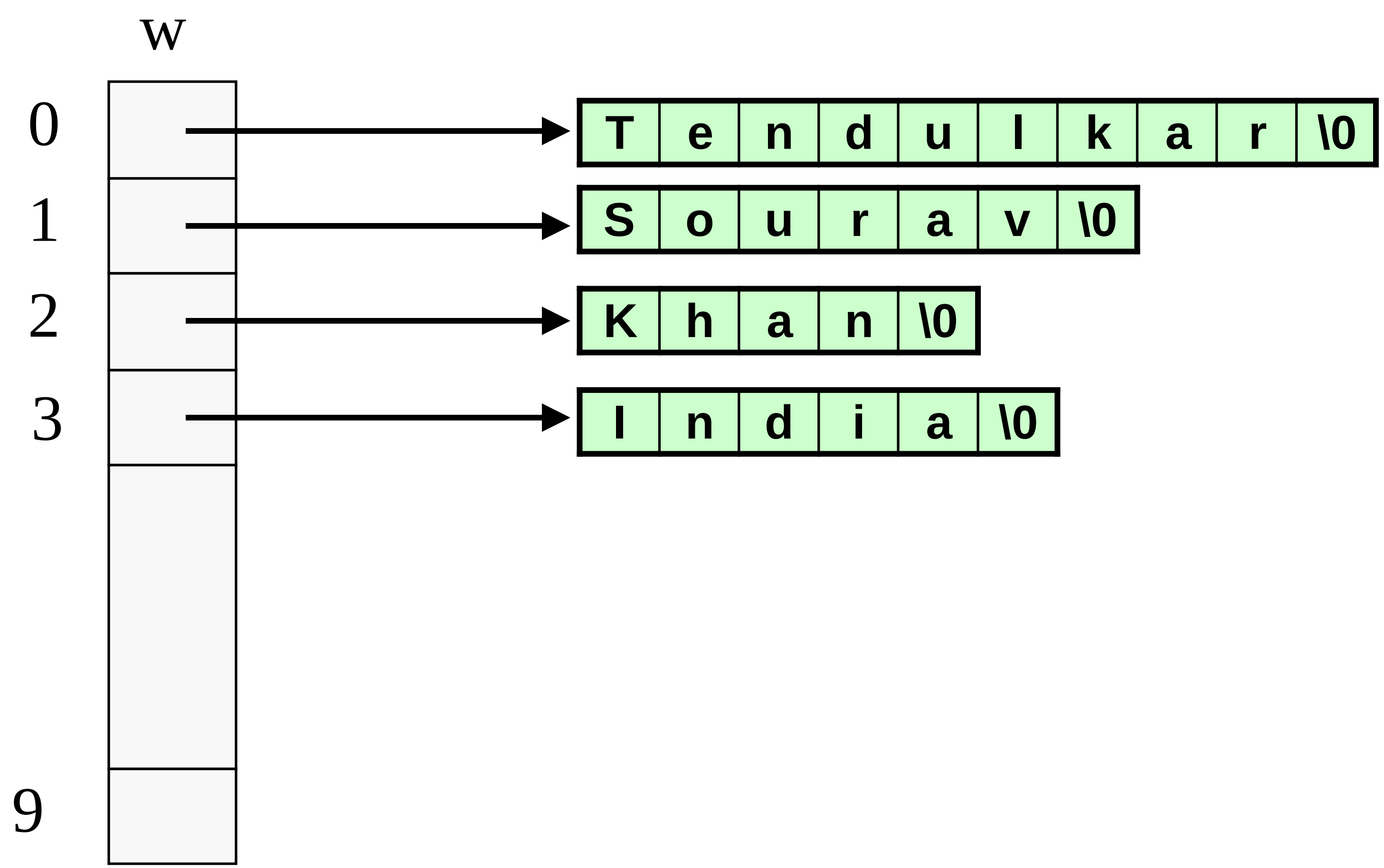
Static array of pointers

```
#define  N   20
#define  M  10
int main()
{
    char word[N], *w[M];
    int i, n;
    scanf("%d",&n);
    for (i=0; i<n; ++i) {
        scanf("%s",  word);
        w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
        strcpy (w[i], word) ;
    }
    for (i=0; i<n; i++) printf("w[%d]  = %s  \n",i,w[i]);
    return 0;
}
```

Output

```
4
Tendulkar
Sourav
Khan
India
w[0] = Tendulkar
w[1] = Sourav
w[2] = Khan
w[3] = India
```

How it will look like



Pointers to pointers

Pointers are also variables (storing addresses), so they have a memory location, so they also have an address

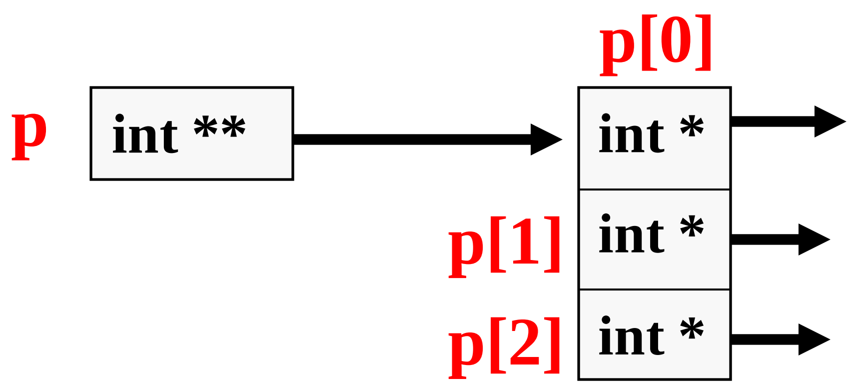
Pointer to pointer – stores the address of a pointer variable

```
int x = 10, *p, **q;
p = &x;
q = &p;
printf("%d  %d  %d", x, *p, *(*q));

will print 10  10  10 (since *q = p)
```

Allocating pointer to pointer

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



Dynamic arrays of pointers

```
int main()
{
  char word[20], **w;  /*  "**w" is a pointer to a pointer array */
  int i, n;
  scanf("%d",&n);
  w = (char **) malloc (n * sizeof(char *));
  for (i=0; i<n; ++i) {
    scanf("%s", word);
    w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
    strcpy (w[i], word) ;
  }
  for (i=0; i<n; i++) printf("w[%d] = %s \n",i, w[i]);
  return 0;
}
```

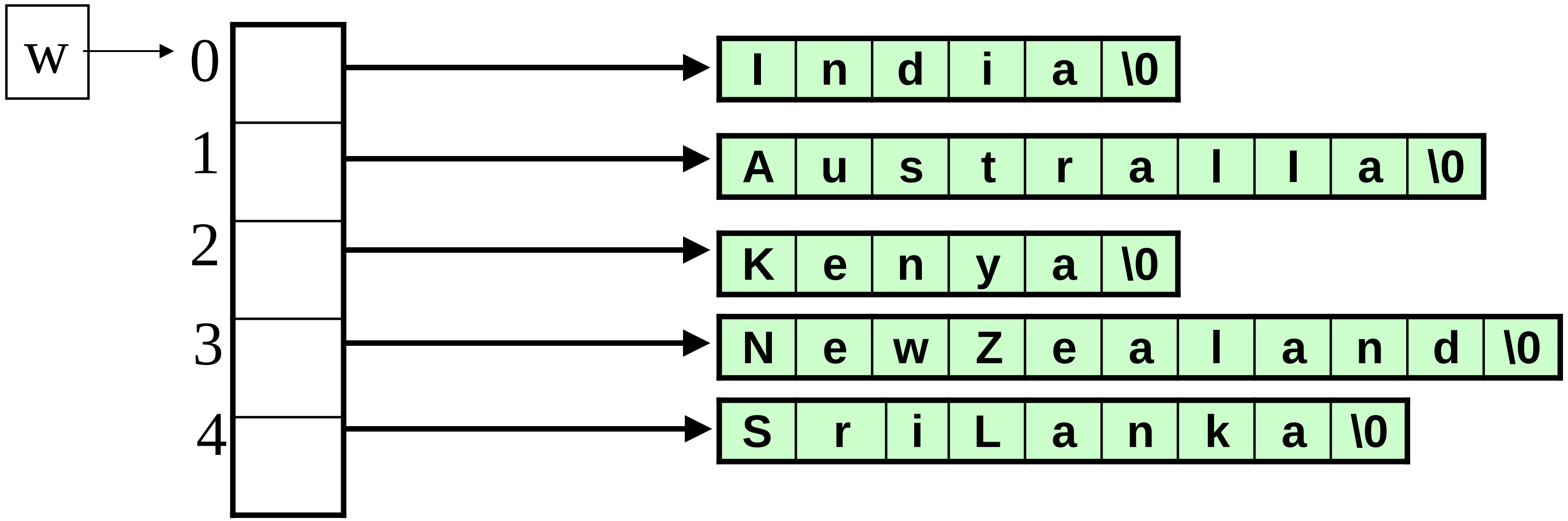
Dynamic arrays of pointers

```
int main()
{
  char word[20], **w;  /*  "**w" is a pointer to a pointer array */
  int i, n;
  scanf("%d",&n);
  w = (char **) malloc (n * sizeof(char *));
  for (i=0; i<n; ++i) {
     scanf("%s", word);
     w[i] = (char *) malloc ((strlen(word)+1)*sizeof(char));
     strcpy (w[i], word) ;
  }
  for (i=0; i<n; i++) printf("w[%d] = %s \n",i, w[i]);
  return 0;
}
```

Output

```
5
India
Australia
Kenya
NewZealand
SriLanka
w[0] = India
w[1] = Australia
w[2] = Kenya
w[3] = NewZealand
w[4] = SriLanka
```

How this will look like



Dynamic allocation of 2-D Arrays

```
int **allocate (int h, int w)
  {
    int **p;
    int i, j;

    p = (int **) malloc(h*sizeof (int *) );
    for (i=0;i<h;i++)
      p[i] = (int *) malloc(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.

Dynamic allocation of 2-D Arrays

```
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");
    }
}
```

```
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 ("\nThe array read as \n");
 print_data (p, M, N);
 return 0;
}
```

Dynamic allocation of 2-D Arrays

```
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
```

```
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 ("\nThe array read as \n");
  print_data (p, M, N);
  return 0;
}
```

Memory layout in dynamic allocation

```
int main()
{
  int **p;
  int M, N;
  printf ("Give M and N \n");
  scanf ("%d%d", &M, &N);
  p = allocate (M, N);
  for (i=0;i<M;i++) {
        for (j=0;j<N;j++)
            printf ("%10d", &p[i][j]);
        printf("\n");
  }
  return 0;
}
```

```
int **allocate (int h, int w)
{
     int **p;
     int i, j;

     p = (int **)malloc(h*sizeof (int *));
     for (i=0; i<h; i++)
                 printf("%10d", &p[i]);
     printf("\n\n");
     for (i=0;i<h;i++)
       p[i] = (int *)malloc(w*sizeof(int));
     return(p);
}
```

Output

```
3 3
31535120  31535128  31535136


31535152  31535156  31535160
31535184  31535188  31535192
31535216  31535220  31535224
```

Starting address of each row, contiguous (pointers are 8 bytes long)

Elements in each row are contiguous

Practice problems

Take any of the problems you have done so far using 1-d arrays or 2-d arrays. Now do them by allocating the arrays dynamically first instead of declaring then statically