

Pointers to pointers & multi-dimensional arrays

Pointers to pointers

malloc function for Dynamic Memory Allocation

```
void *malloc(size_t size);
```

- **size_t** – represents **unsigned** (non-negative values) of **integer**, **char**, **long**, **long long**, **short**.
- **malloc()** - allocates size consecutive bytes in memory and returns the address of the first byte, a pointer to it.
- **void*** - **if malloc() does not know how we are going to use this piece of memory** (array of chars? array of doubles? etc.), so it returns the memory as a **general pointer**
- The allocated memory is placed on the heap.
- The allocated memory contains 'garbage'.

Calloc function for Dynamic Memory Allocation

```
void *calloc(size_t nitems, size_t size);
```

- **calloc** does the same thing as **malloc**, but:
 - It gets two parameters instead of one.
 - The memory is 'cleaned', all the bytes are 0.
- Both **malloc** and **calloc** return **NULL** if there is insufficient memory on the heap.

Thus, it is critical to check if NULL.

Library Function free()

```
void free(void *ptr);
```

- **free()** - frees the block of memory that was allocated with **malloc()** or **calloc()**.
- Why should we use free ?...
 - The memory allocated on the heap is not automatically released, which can cause 'memory leak'.
 - The heap is not endless. If, for example, a function performs many dynamic allocations, why not release memory when it is no longer needed to make room for new allocations?

free() – cont'd

Question :

- How does the computer know how many bytes to free ?

Answer :

- The operating system manages a table of addresses.
- When we allocate memory, the address of the first byte is written in that table, as well as the number of bytes that were allocated.
- When we call the function **free()**, we send a pointer to the first byte of the allocated memory. The operating system finds that address in the table and sees how many bytes to free.

Reminder – the swap function

Does nothing

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
int main()
{
    int x, y;
    x = 3; y = 7;
    swap(x, y);
    // now x==3, y==7
}
```

Works

```
void swap(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
```

```
int main()
{
    int x, y;
    x = 3; y = 7;
    swap(&x, &y);
    // x == 7, y == 3
}
```

Pointers to pointers (double pointer)

- Just to remind -

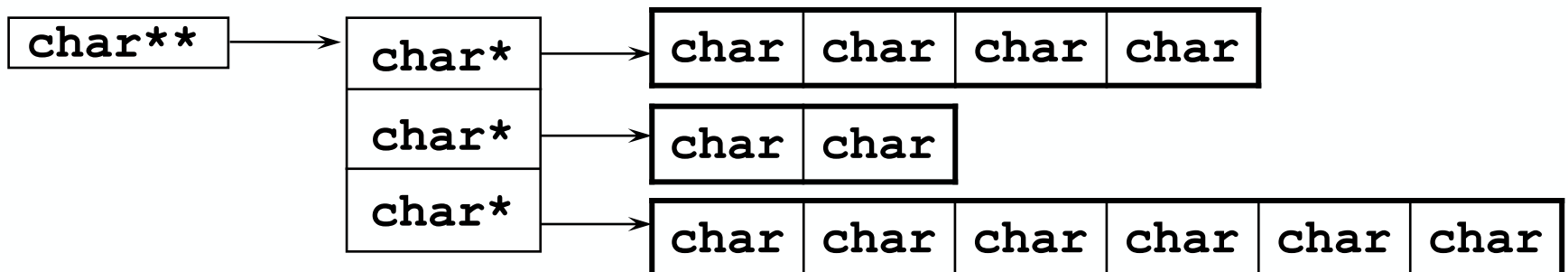
`p` Is a pointer to an integer. It is stored in an address and it also stores an address.

Printing `p` will print the address `p` points to.

Printing `*p` prints the value `p` points to.

Printing `&p` prints the address of `p`, which is unique to `p` and was allocated in the moment it was declared.

names



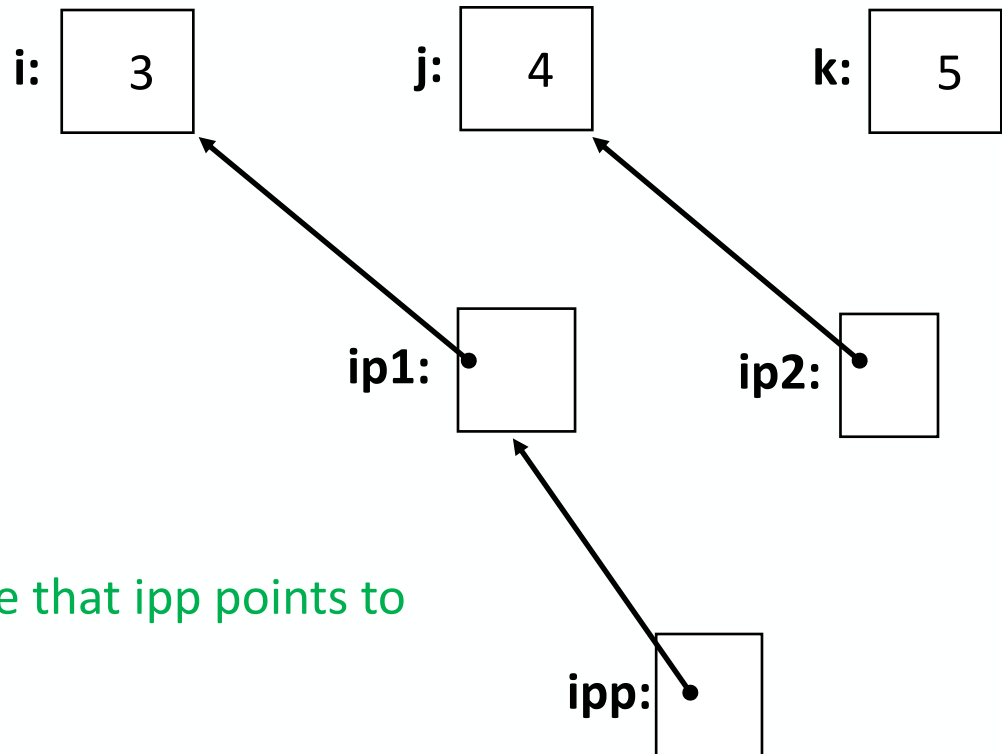
Pointers to pointers (double pointer)

```
int i=3;  
int j=4;  
int k=5;
```

```
int *ip1 = &i;  
int *ip2 = &j;  
int **ipp = &ip1;
```

```
*ipp=ip1 // the value that ipp points to
```

```
**ipp=i
```



Pointers to pointers

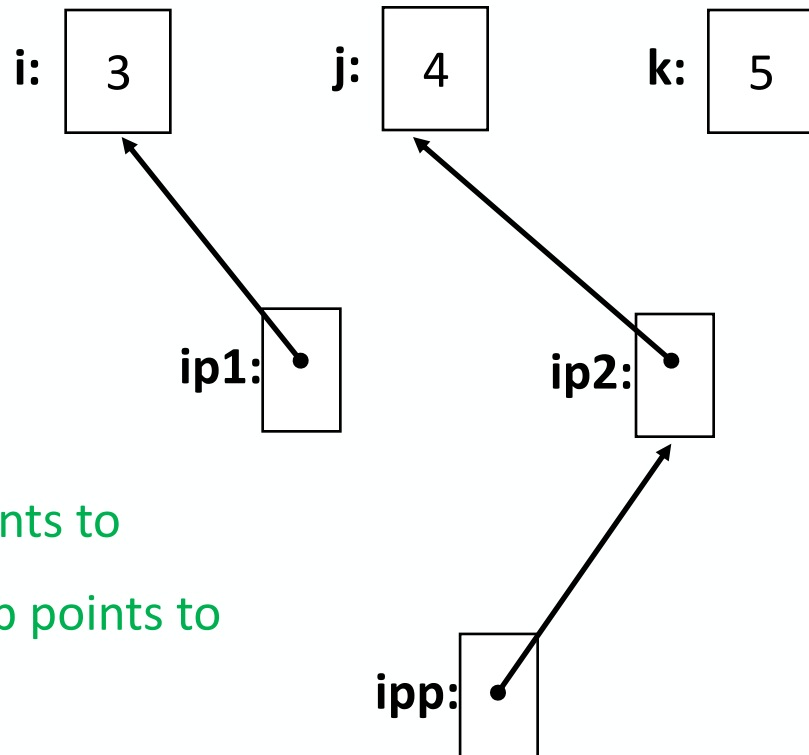
```
int i=3;  
int j=4;  
int k=5;
```

```
int *ip1 = &i;
```

```
int *ip2 = &j;
```

```
*ipp=ip2 // the value that ip2 points to
```

```
ipp = &ip2; // the address that ip2 points to
```



Pointers to pointers

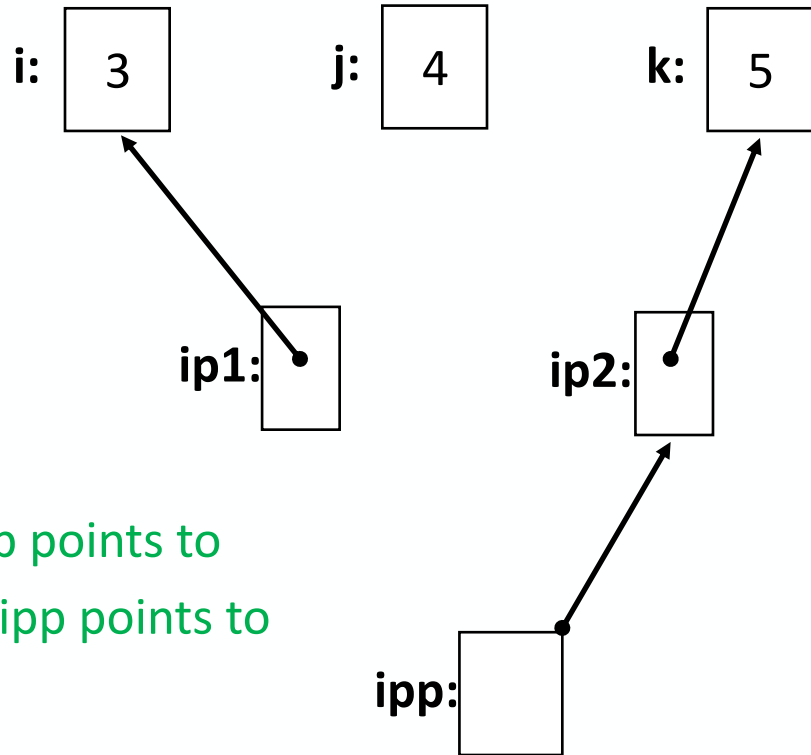
```
int i=3;  
int j=4;  
int k=5;
```

```
int *ip1 = &i;
```

```
int *ip2 = &j;
```

```
ipp = &ip2; // the address that ipp points to
```

```
*ipp = ip2 = &k; // the value that ipp points to
```



Strings / Arrays are POINTERS

Pointers to pointers: example

//put pointer to an allocated string in pp

```
int allocString(size_t len, char ** pp)
{
    char *str = (char*)malloc(len + 1);
    if (str==NULL)
    {
        return 1;
    }
    *pp = str;
    return 0;
}
```

Pointers to pointers: example

```
// copy a string using "allocString"
```

```
void main()
```

```
{
```

```
    char *s = "example";
```

```
    char *copy;
```

```
    allocString(strlen(s), &copy);
```

```
    strcpy(copy, s);
```

```
    free (copy);
```

```
}
```

Array of pointers – grades.c - exercise

- 1 `/* grades.c`
- 2 This program calculates the average grade of a class. The
- 3 programmer doesn't know the number of students in the class, so
- 4 he can't declare an array. Dynamic allocation is used instead */

Array of pointers – grades.c - solution

```
1  /* grades.c
2  This program calculates the average grade of a class. The
3  programmer doesn't know the number of students in the class, so
4  he can't declare an array. Dynamic allocation is used instead */
5
6  #include <stdio.h>
7  #include <stdlib.h> /* for malloc, free and exit */
8  #define EXIT_FAIL 42
9
10 void main(void)
11 {
12     float *grades = NULL; /* the grades are of type float */
13     int num_students; /* will be entered by the user */
14     float sum = 0; /* will help calculating the average. */
15     int ctr;
```


Array of pointers – grades.c - solution

```
18     do
19         { /* loop until a legal number of students is entered */
20             printf("How many students are there in class?=> ");
21             scanf("%d", &num_students);
22         } while(num_students <= 0);
23         /* now we have the number of students, so we can dynamically
24         allocate an array for holding their grades (floats) */
25         grades = (float*)malloc(num_students*sizeof(float));
26         /* if there isn't enough consecutive memory, malloc() returns NULL */
27         if(grades=NULL)
28         {
29             puts("There is not enough memory. Sorry.\n");
30             exit (EXIT_FAIL); /* exit the program */
31         }
```

Array of pointers – grades.c - solution

```
34     for (ctr = 0; ctr < num_students ; ++ctr) /* get the grades */
35     {
36         printf("Enter grade #%d: ", ctr + 1);
37         scanf("%f", &grades[ctr]);
38         sum += grades[ctr]; /* for calculating the average later */
39     }
40     printf("The average of the grades is : %f\n", sum/num_students);
41     /* memory that was allocated by malloc/calloc must be freed! */
42     free(grades); /* free the allocated memory */
43 }
```

Array of pointers

The problem:

- We have to read, for example, a list of 10 names.
- The names differ in length.
- The maximum length is, in our case, 23.
- First Solution:

```
char arr[10][23];
```

A list of names is an array of strings. Since a string is an array of char, we have an array of arrays of char - a two-dimensional array of char. Look at the **wasted memory !**

[illegible]

Array of pointers

- Second Solution:

```
char *arr[10];
```

arr is array of 10 elements, each one of them is a pointer to string/array of chars. Each element will point to a different array of a different size.

[illegible]

Example: names1.c

```
1  /*names1.c Creates an array of strings, each of an unknown length*/
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #define EXIT_FAIL 42
6  void free_names(char * names[], int num); /* function prototype */
7
8  int get_names(char *names[], int num)
9  {
10     int i = 0;
11     char str[128];
12     do {
13         puts("Enter a name (press ENTER to stop) => ");
14         if (gets(str) == NULL)
15             break;
16
```

Example: names1.c

```
18         if( (names[i] = (char*) malloc(strlen(str)+1)) == NULL)
19             {
20                 puts ("Not enough memory. Sorry.\n");
21                 free_names(names, i); /* some memory has been allocated ! */
22                 exit(EXIT_FAIL);
23             }
24         strcpy(names[i], str); /* copy the name into the memory */
25     }while (++i < num);
26     return i; /* return the actual number of names in the array */
27 }
28 void print_names(const char *names[], int num) {
29     int i;
30     for (i = 0 ; i < num ; i++)
31         printf("%2d) %s\n", i + 1, names[i]);
32 }
```

Example: names1.c

```
37 void free_names(char * names[], int num)
38 {
39     int i;
40     for( i = 0 ; i < num ; ++i )
41         free(names[i]);
42 }
43
44 void main(void)
45 {
46     char *names[10]; /* an array of 10 pointers to char */
47     int final_num; /* in case the user enters less than 10 names */
48     final_num = get_names(names, 10); /* get names into the array */
49     print_names(names, final_num); /* print the array of names */
50     free_names(names, final_num); /* free all the allocated memory */
51 }
```

Super Dynamic Allocation

- The previous program allocated memory for the names according to their lengths. No memory was wasted in the case of short names, and no memory was missing in the case of long names.
- BUT it is good only as long as the user doesn't have more than 10 friends' names to keep. If the user has 11 friends, he must throw one name away. How can he choose who to insult ?...
- The next program will not only allocate memory for the names according to their lengths, but it will also allocate the array itself according to the number of names that the user wants to keep !
- Turn to the next page for SUPER DYNAMIC ALLOCATION !

Example: names2.c

- 1 `/* names2.c - Creates an array for unknown number of strings. The length of each string is also unknown. Dynamic memory allocation is used. */`

Example: names2.c - solution

```
1  /* names2.c - Creates an array for unknown number of strings. The length of each string is also
   unknown. Dynamic memory allocation is used. */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #define EXIT_FAIL 42
6  void free_names(char **names, int num)
7  {
8      while(num > 0) {
9          free(names[--num]);
10     }
11 }
```

Example: names2.c - solution

```
1 void print_names(const char **names, int num) {
2     int ctr;
3     printf("The following %d names were entered :\n", num);
4     for(ctr = 0 ; ctr < num ; ++ctr)
5         printf("%2d) %s\n", ctr + 1, names[ctr]);
6 }

22 void get_names(char **names, int *num) {
23     int i = 0;
24     char str[128]; /* for getting names from the user */
25     /* loop until num names have been entered or until the user chooses to stop entering names */
26     do {
27         printf("Enter the #%d name (press CTRL/Z ENTER to stop)=> ", i+1);
28         if(gets(str) == NULL) /* the user wants to stop entering names */
29             break;
30         if((names[i] = (char*)malloc(strlen(str)+1))==NULL) {
31             puts ("Not enough memory. Sorry.\n");
32             free_names(names, i); /* some memory was allocated! Free it ! */
33             free(names);
34         }
35         exit(EXIT_FAIL);
36     }
```

Example: names2.c - solution

```
43         strcpy(names[i], str); /* copy the name into the memory */
44     }while (++i < *num);
45     *num = i; /* update the number of names */
46 }
47 int get_number_of_names()
48 {
49     int num = 0;
50     while(num <= 0) /* loop until a legal number has been entered */
51     {
52         printf("How many names do you wish to enter ? => ");
53         scanf("%d", &num);
54         getchar();
55     }
56     return num;
57 }
```

Example: names2.c - solution

```
64 void main(void)
65 {
66     char **names = NULL; /* will be an array of string */
67     int num; /* the number of names that the user wants to keep */
68     num = get_number_of_names();
69     /* allocate memory for the number of users, each is a pointer to char */
70     names = (char**)malloc(num * sizeof(char*));
71     if(! names) /* there is not enough available consecutive memory */
72     {
73         puts("There is not enough memory for the program. Sorry.\n");
74         exit(EXIT_FAIL);
75     }
76     /* call a function that will get the names. The variable num is sent by address so that if the
77     user chooses to enter fewer names that he originally intended to enter, the function will
78     change num to the correct number of names that were entered.*/
```

Example: names2.c - solution

```
85  get_names(names, &num);
86
87      /* print the array of names */
88      print_names(names, num);
89
90      /* free the memory that was dynamically allocated for the names */
91      free_names(names, num);
92
93      /* the array of pointers was also allocated dynamically. Free it */
94      free(names);
95  }
```

Pointers to pointers to pointers ...

We also have pointers to pointers to pointers, etc.:

```
double ** mat1 = getMatrix();
```

```
double ** mat2 = getMatrix();
```

```
//allocate an array of matrices
```

```
double *** matrices = (double***) malloc(n*sizeof(double**));
```

```
matrices[0] = mat1;
```

```
matrices[1] = mat2;
```

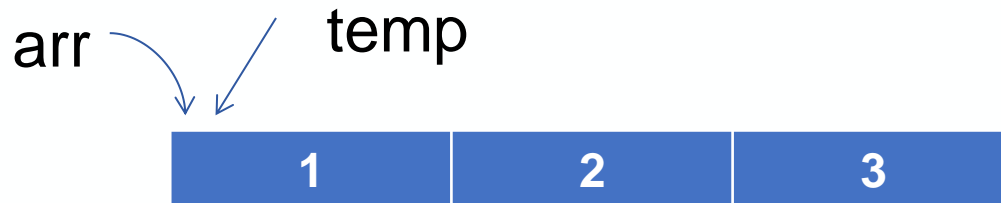
Array reallocation

Allocate array on the heap and assign a pointer to it



Array reallocation

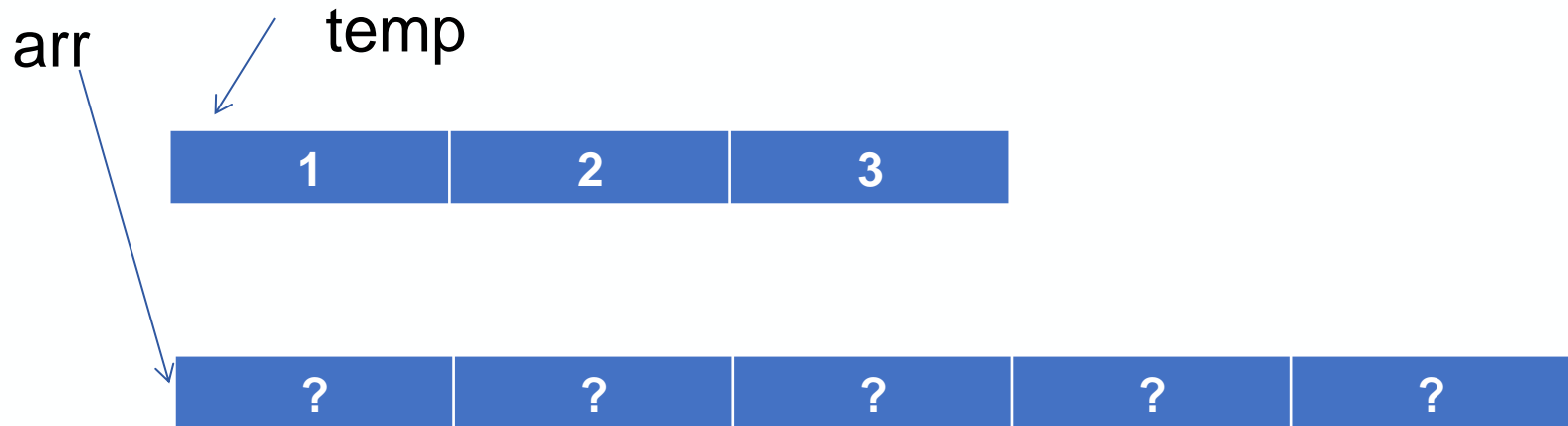
Use temporary pointer to point to the old array.



Array reallocation

Use temporary pointer to point to the old array.

Reallocate new array.

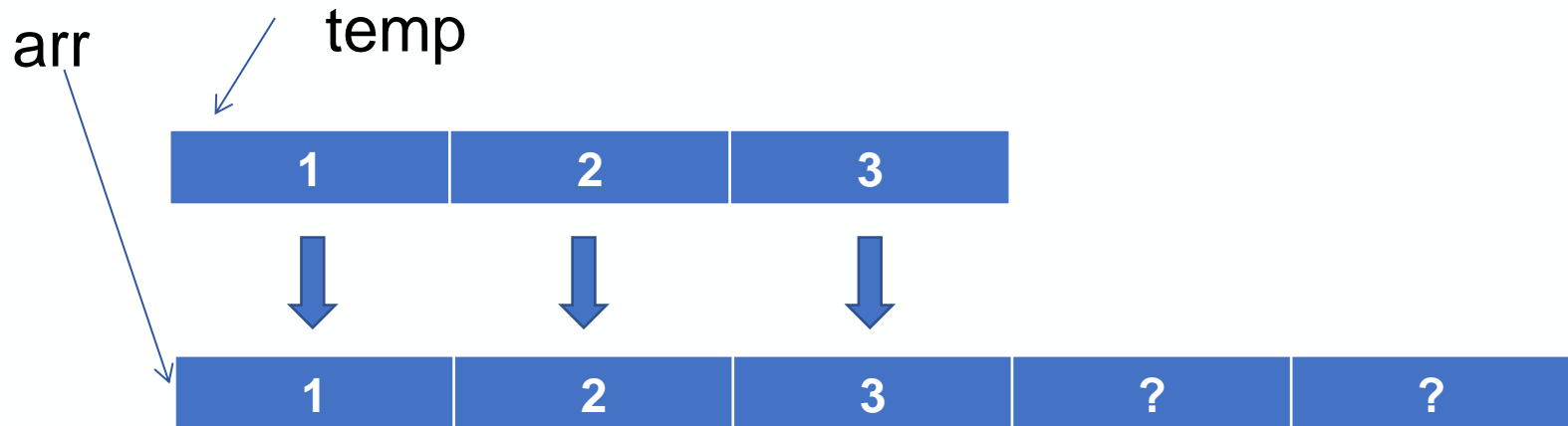


Array reallocation

Use temporary pointer to point to the old array.

Reallocate new array.

Copy values from the old array to the new one.



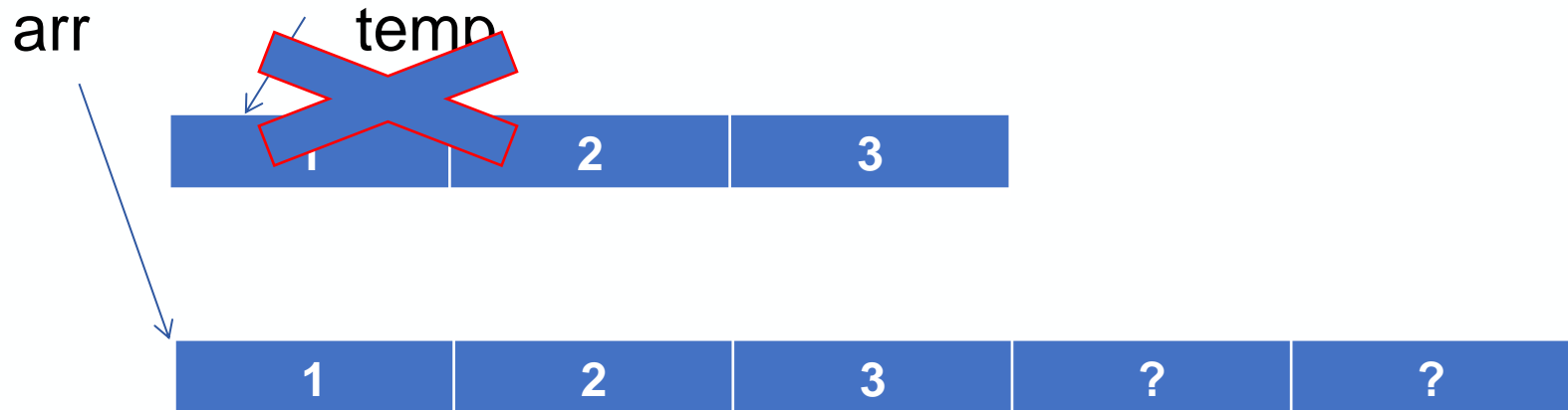
Array reallocation

Use temporary pointer to point to the old array.

Reallocate new array.

Copy values from the old array to the new one.

Free the old array.



Array reallocation – version 1 (new pointer to an array)

```
void reallocateMemory(int **parr, size_t oldSize, size_t newSize)
{
    int *old = *parr;

    //partial example—do not forget checking malloc!

    *parr = (int*)malloc(sizeof(int)*newSize);
    size_t i = 0;
    while(i < oldSize) {
        (*arr)[i] = old[i];
        ++i;
    }
    free(old);
}
```

Array reallocation – version 1

```
int main() {  
  
    int *arr = (int*)malloc(sizeof(int)*arrSize);  
  
    //do some stuff  
  
    ...  
  
    reallocateMemory(&arr,arrSize,newSize);  
  
    free(arr)  
}
```

Array reallocation – version 2 (array is like a pointer)

```
int* reallocateMemory(int *arr, size_t oldSize, size_t newSize)
{
    int *old = arr;

    //partial example—do not forget checking malloc!
    arr = (int*)malloc(sizeof(int)*newSize);
    size_t i = 0;
    while(i < oldSize) {
        arr[i] = old[i];
        ++i;
    }
    free(old);
    return arr;
}
```

Array reallocation – version 2

```
int main() {  
  
    int *arr = (int*)malloc(sizeof(int)*arrSize);  
  
    //do some stuff  
  
    ...  
    arr = reallocateMemory(arr,arrSize,newSize);  
    free(arr);  
}
```


Array reallocation – using realloc

```
int * arr = (int*)malloc(sizeof(int)*oldSize);
```

```
arr = (int*)realloc(arr,sizeof(int)*newSize);
```

realloc tries to reallocate the new memory in place,

if fails, tries elsewhere

The old data is preserved

Multi-dimensional arrays

Multi-dimensional arrays

Can be created in few ways:

1. Automatically:

```
int m[2][3]; // 2 rows, 3 columns
```

- Continuous memory
- Access: `m[row][col] = 0;`

2. Semi-dynamic (array of pointers):

```
int* m[5]; // allocates memory for 5 pointers
```

```
for (i=0; i<5; ++i)
```

```
{
```

```
    m[i] = (int*) malloc( 7*sizeof(int) ); // m[i] now points to a memory for 7 ints
```

```
}
```

3. Dynamically (double-pointers):

```
int ** m;
```

```
m = (int**) malloc( 5*sizeof(int*) );
```

```
for (i=0; i<5; ++i) {
```

```
    m[i] = (int*)malloc( 7*sizeof(int) );
```

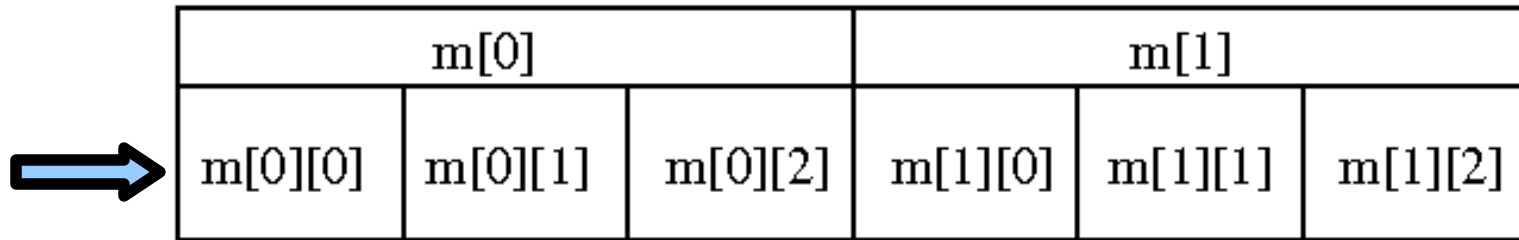
```
}
```

Automatic multi-dimensional arrays

```
#define R 2
```

```
#define C 3
```

```
int m[R][C];
```



```
int* m_ptr;
```

```
for (m_ptr= &(m[0][0]);
```

```
    m_ptr<=&(m[R-1][C-1]);
```

```
    ++m_ptr) {
```

```
    printf("%d\n", *m_ptr );
```

```
}
```

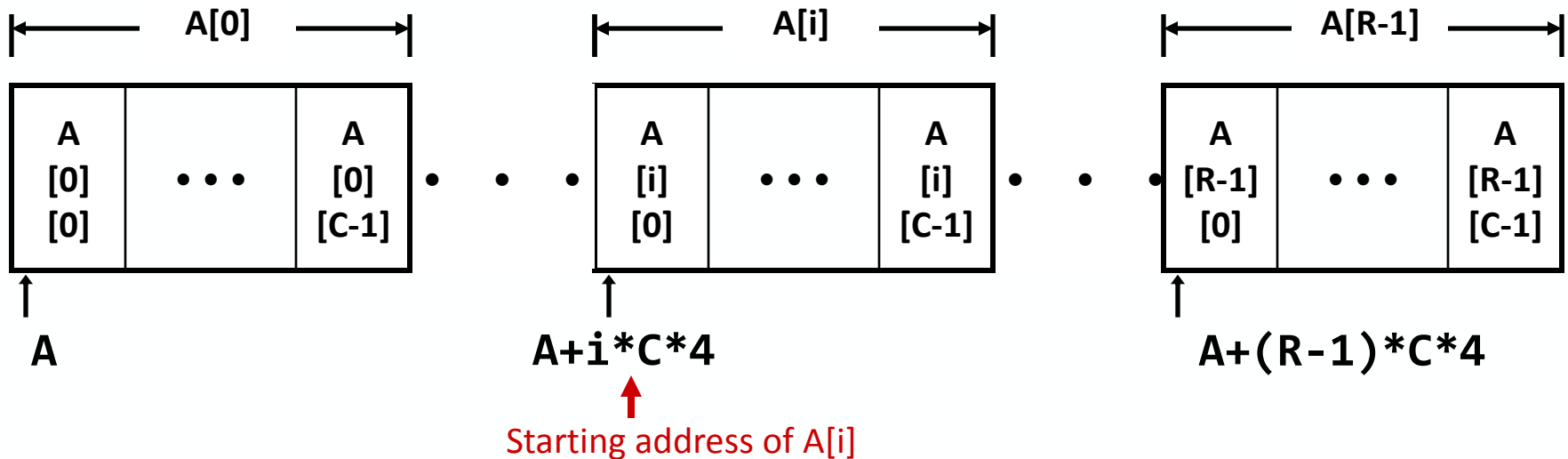
Automatic multi-dimensional arrays

int A[R][C];

Like a matrix →

A[0][0]	A[0][1]	...	A[0][C-1]
...
A[i][0]	A[i][1]	...	A[i][C-1]
...
A[R-1][0]	A[R-1][1]	...	A[R-1][C-1]

Row-major ordering →



Semi-dynamic multi-dimensional arrays

2. Semi-dynamic:

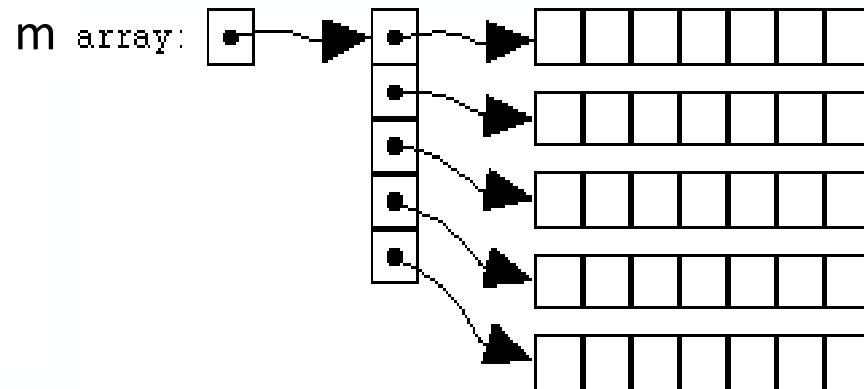
Define an array of pointers:

```
int* m[5]; // allocates memory for 5 pointers
for (i=0; i<5; ++i)
{
    m[i] = (int*) malloc( 7*sizeof(int) ); // m[i] now points to a memory for 7 ints
}
```

Dynamic multi-dimensional arrays

3. Dynamically:

```
int ** m;  
m = (int**) malloc( 5*sizeof(int*) );  
for (i=0; i<5; ++i)  
{  
    m[i] = (int*)malloc( 7*sizeof(int) );  
}
```



Dynamic Allocation of Multi-dimensional array – summary

- Following are different ways to create a 2D array on heap (dynamically allocate a 2D array). In the following examples, we have considered 'r' as number of rows, 'c' as number of columns and we created a 2D array with $r = 3$, $c = 4$ and following values

- Using a single pointer:

A simple way is to allocate memory block of size $r*c$ and access elements using simple pointer arithmetic.

- Using an array of pointers:

We can create an **array of pointers of size r**. After creating an array of pointers, we can dynamically allocate memory for every row.

- Using pointer to a pointer (more flexible than an array of pointers):

We can create an array of pointers (any size so more flexible) also dynamically using a double pointer. Once we have an array - pointers allocated dynamically, we can dynamically allocate memory and for every row like method 2

Dynamic Allocation of 2D array – Single Pointer

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

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

int main()
{
    int r = 3, c = 4;
    int *arr = (int *)malloc(r * c * sizeof(int));

    int i, j, count = 0;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            *(arr + i*c + j) = ++count;

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", *(arr + i*c + j));

    /* Code for further processing and free the
       dynamically allocated memory */


    return 0;
}
```

Output:

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

Dynamic Allocation of 2D array – Array of Pointers

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



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

int main()
{
    int r = 3, c = 4, i, j, count;

    int *arr[r];
    for (i=0; i<r; i++)
        arr[i] = (int *)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as (*(arr+i)+j)
    count = 0;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count; // Or (*(arr+i)+j) = ++count

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);

    /* Code for further processing and free the
       dynamically allocated memory */

    return 0;
}
```

Output:

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

Dynamic Allocation of 2D array – Pointer to Pointer with two mallocs

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

int main()
{
    int r = 3, c = 4, i, j, count;

    int **arr = (int **)malloc(r * sizeof(int *));
    for (i=0; i<r; i++)
        arr[i] = (int *)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as (*(arr+i)+j)
    count = 0;
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count; // OR (*(arr+i)+j) = ++count

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);

    /* Code for further processing and free the
       dynamically allocated memory */

    return 0;
}
```

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

Output:

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

Dynamic Allocation of 2D array –

Double Pointer and one malloc



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



```
int main()
{
    int r=3, c=4, len=0;
    int *ptr, **arr;
    int count = 0,i,j;
```



```
    len = sizeof(int *) * r + sizeof(int) * c * r;
    arr = (int **)malloc(len);
```

```
    // ptr is now pointing to the first element in of 2D array
    ptr = (int *)(arr + r);
```

```
    // for loop to point rows pointer to appropriate location in 2D array
```

```
    for(i = 0; i < r; i++)
        arr[i] = (ptr + c * i);
```

```
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            arr[i][j] = ++count; // OR (*(arr+i)+j) = ++count
```

```
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            printf("%d ", arr[i][j]);
```

```
    return 0;
```

```
}
```

```
<
```

Output:

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

Passing Multi-dimensional arrays to functions

- One important thing for passing multidimensional arrays is, **first dimension does not have to be specified**. The **second dimensions must be given**
- So, there are few options to pass N-D array to a function
 - When **both dimensions are available** (either as a macro or as a global constant).
 - When **only second dimension is available** (either as a macro or as a global constant).
 - **Using a pointer**

Passing 2D array – both dimensions available

```
#include <stdio.h>
const int M = 3;
const int N = 3;

void print(int arr[M][N])
{
    int i, j;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr);
    return 0;
}
```

Passing 2D array – 2nd dimension is available

```
#include <stdio.h>
const int N = 3;

void print(int arr[][N], int m)
{
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < N; j++)
            printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr, 3);
    return 0;
}
```

Passing 2D array – single pointer

```
#include <stdio.h>
void print(int *arr, int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            printf("%d ", *((arr+i*n) + j));
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;

    // We can also use "print(&arr[0][0], m, n);"
    print((int *)arr, m, n);
    return 0;
}
```


Passing Multi-dimensional arrays to functions - summary

```
int x[5][7]; // 5 rows, 7 columns
```

When sending 'x' as an argument to a function, only the 1st index can be omitted:

- `void func(int x[5][7]) // ok`
- `void func(int x[][7]) // ok`
- `void func(int x[][]) // does not compile`
- `void func(int * x[]) // by a single pointer`
- `void func(int ** x) // by pointer to a pointer`

Array of Pointers Vs. Pointer to an Array

- **()** means a lot
- **int (*m)[5]** declares m as a pointer to an array of 5 integers
- **int *m[5]** declares m as an array of 5 pointers, each pointer points to integer same as **int *(m[5])**

1- use of *m[5]:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(void){
4     int *m[5];
5     int i;
6     for(i=0;i<5;++i)
7         m[i] = (int *)malloc(sizeof(int));
8     for(i=0;i<5;++i)
9         *m[i]=2*i+1;
10    for(i=0;i<5;++i)
11        printf("%d\n", *m[i]);
12    for(i=0;i<5;++i)
13        free(m[i]);
14    return 0;
15 }
```

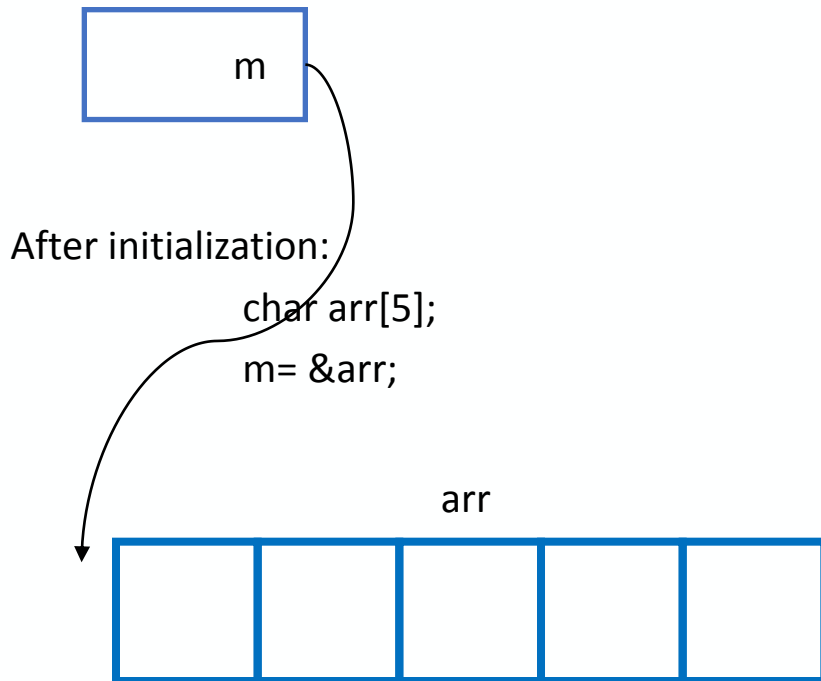
2-use of (*m)[5]:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void){
5     int (*m)[5];
6     int a[5]={1,2,3,4,5};
7     m=&a;
8     int i;
9     for(i=0;i<5;++i)
10        printf("%d\n", (*m)[i]);
11    return 0 ;
12 }
```

Pointer to Array Vs. Array of Pointers

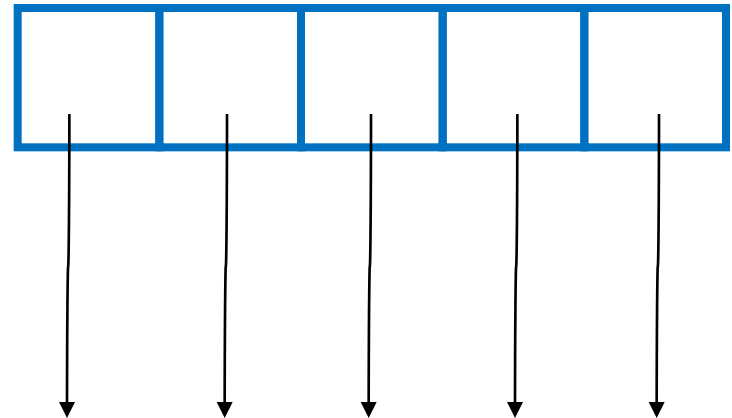
```
char (*m)[5];
```

Pointer to an array of 5 chars



```
char *m[5];
```

Array of 5 pointers



Command line arguments: argv, argc

- The main() function is mostly defined with a return type of int and without parameters.
- We can also give command-line arguments.
- Command-line arguments are given after the name of the program in command-line shell of Operating Systems (e.g. linux command line).
- To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments (argc) and second is list of command-line arguments (argv).

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

Command line arguments: argv, argc

Properties of Command Line Arguments:

- They are passed to main() function.
- They are parameters/arguments supplied to the program when it is invoked.
- They are used to control program from outside instead of hard coding those values inside the code.
- argv[argc] is a NULL pointer.
- argv[0] holds the name of the program.
- argv[1] points to the first command line argument and argv[n] points last argument.

Command line arguments: argv, argc

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name).

The value of argc should be non negative.

- **argv (ARGument Vector)** is array of character pointers listing all the arguments.

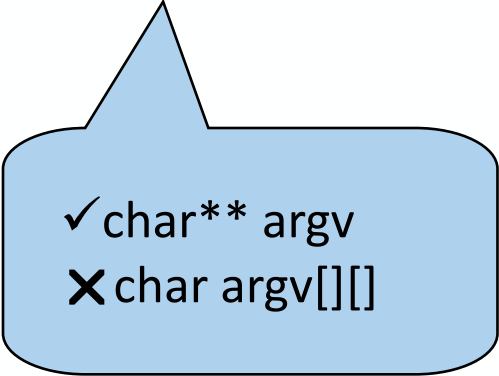
If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.

Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

Command line arguments: argv, argc

- To pass command line arguments to our program we should use the following **main** declaration:

```
main(int argc, char* argv[]) { ...
```



✓ char** argv
✗ char argv[][]

argv & argc: example

```
$ prog1 -u danny -p 1234
```

```
argc == 5
```

```
argv[0] == "prog1"
```

```
argv[1] == "-u"
```

```
...
```

```
argv[4] == "1234"
```

Always: argv[argc] == NULL

(redundant since we are also given argc)