BITS Pilani

Pilani Campus

Module 11 – Dynamic Memory Allocation

Dr. Jagat Sesh Challa

Department of Computer Science & Information Systems

Module Overview

- Stack vs Heap memory
- Dynamic Memory Allocation
- Dynamically allocated arrays
- Memory Management Issues
- Dynamically Allocated Arrays of Structures
- Multi-dimensional arrays using dynamic allocation
- Command Line Arguments

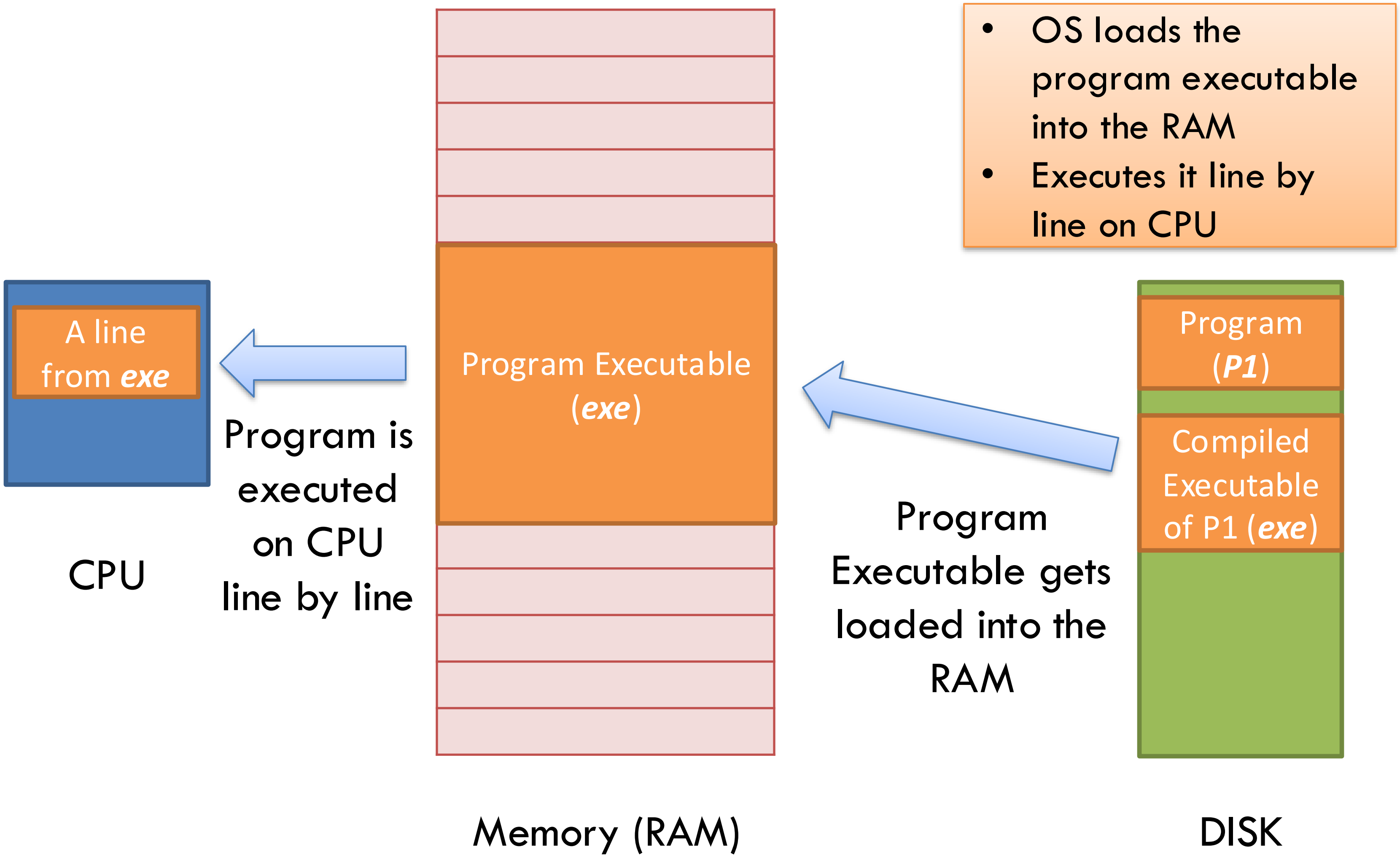
innovate achieve lead

BITS Pilani

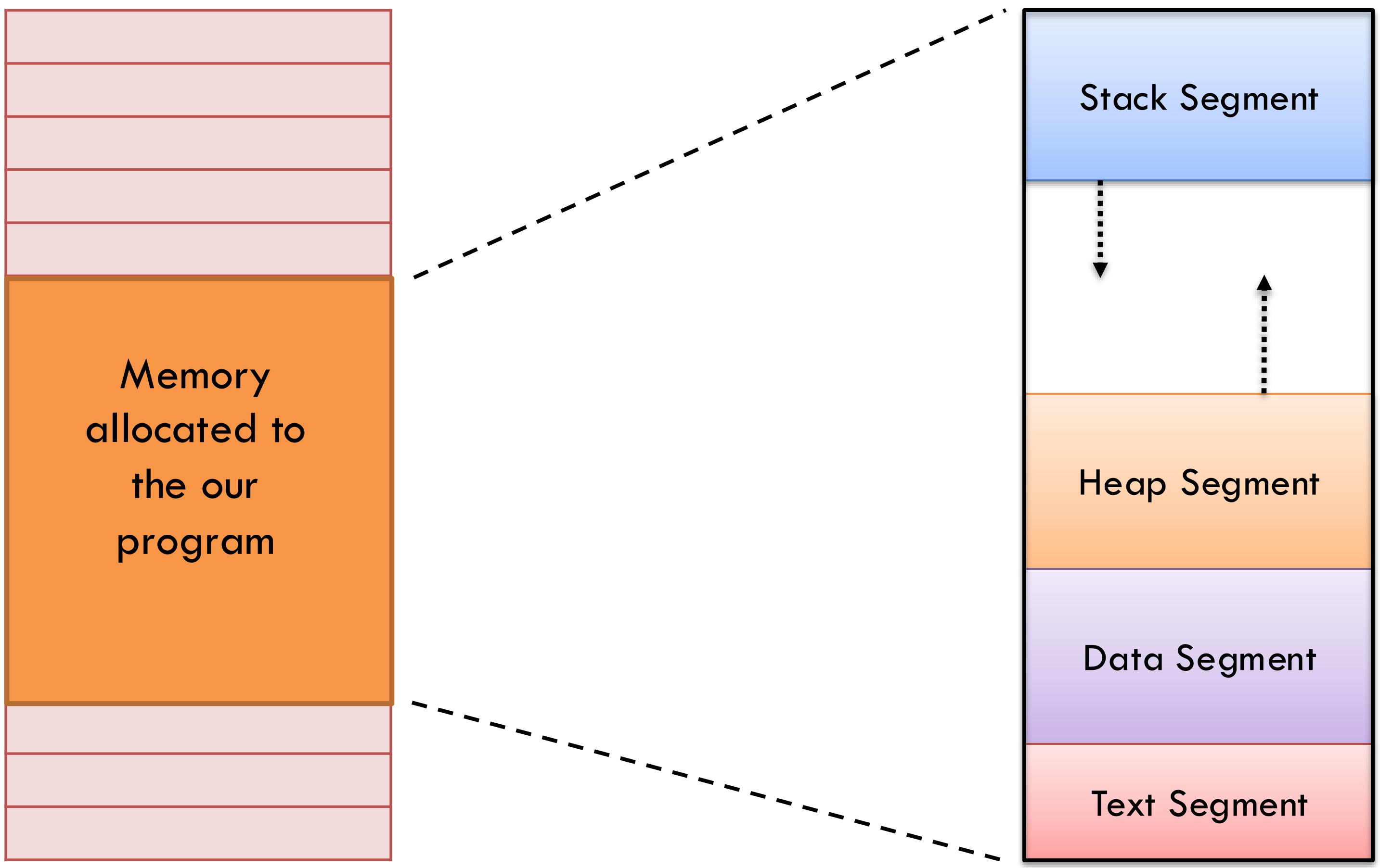
Pilani Campus

Stack vs Heap Memory

Our block diagram is back again!



Looking at the main memory only

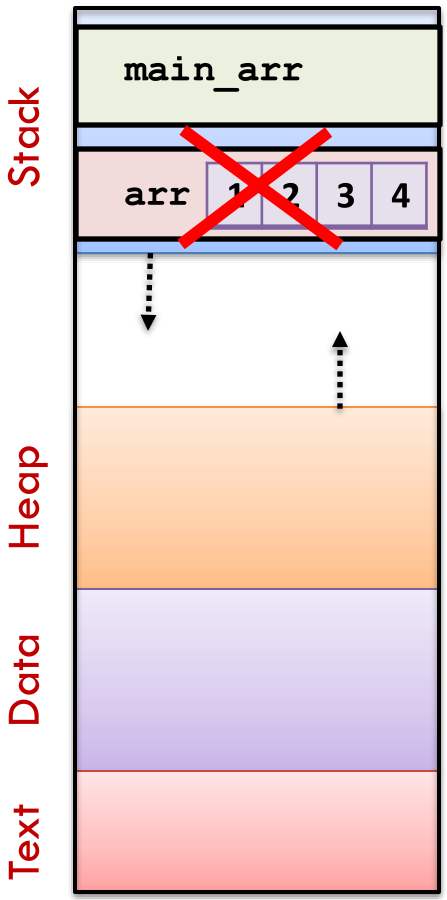


Stack vs Heap

Stack:

- One frame allocated to each function call
- Auto (or local) variable defined inside the frame allocated for a function
- Memory allocated to each frame is recalled after the function execution finishes

Let us try to explain with our memory diagram



Stack frame allocated to *main()* in stack

Stack Frame allocated to *f1()* in the stack

Memory allocated to our program

When f1() returns, the memory allocated to it is destroyed. So arr declared inside f1() does not exist anymore. Accessing arr in main function now gives an error.

Consider this program:

```
int f1(){
    int arr[4] = {1, 2, 3, 4};
    return arr;
}
int main(){
    int main_arr[] = f1();
    printf("First ele is: %d", main_arr[0]);
    return 0;
}
Output:
Error!
```

Advantage of Heap over Stack

Stack:

- One frame allocated to each function calls
- Auto (or local) variable defined inside the frame allocated for a function
- Storage allocated for each frame is recalled after the function execution finishes

Heap:

- Heap is dynamically and explicitly allocated by programmer
- So allocated storage is not recovered on function return
- Programmer has to deallocate (at his/her convenience)

The memory allocated in heap is accessible globally to all functions

innovate achieve lead

BITS Pilani
Pilani Campus

Dynamic Memory Allocation

Motivation

- **Limitation of Arrays**
  - Amount of memory is fixed at compile time and cannot be modified.
    - *We tend to oversize Arrays to accommodate our needs.*
  - We can't return an array from a function as the memory allocated to the function gets destroyed on return.
- **Solution:**
  - Allocate memory dynamically at run time.
    - *The size of the array can be taken as user input*
    - *We can re-allocate the size of the array of it gets full*
  - Array gets memory in heap, hence can be accessed by all functions of the program
- **Flexibility comes with a price:**
  - Programmer should free up memory when no longer required.

Using `malloc()` and `calloc()`

Using `realloc()`

Using `free()`

Allocating memory dynamically – malloc(): Syntax

<stdlib.h> is the header file for using malloc() and other dynamic memory allocation related operations

```
#include <stdlib.h>

void fun1()
{


            int * pt = (int *) malloc (sizeof(int));
}
```

Convert pointer to byte(s) (generic pointer) to the type pointer to int

Ask Operating System to allocate memory

Number of bytes of memory required

Creating a dynamically allocated array

```
#include <stdlib.h>
#define MAX_SIZE 10

void fun2()
{


    int * arr = (int *) malloc( MAX_SIZE * sizeof(int));
}
```

Number of bytes of memory required

malloc()

Observations:

- Library `stdlib` provides a function malloc
- `malloc` accepts number of bytes and returns a pointer (of type void) to a block of memory
- Returned pointer must be "type-cast" (i.e. type converted) to required type before use.
- `malloc` doesn't initialize allocated blocks, each byte has a random value.
- The block of memory returned by `malloc` is allocated in heap

Reading between the lines:

- Number of bytes is dynamic – unlike in arrays where size is constant.
- Can number of bytes be arbitrary?
  - *No! Memory is of finite and fixed size!*

Example

```c
short * p1;
int * p2;
float * p3;
```

Pointer variable declaration

```c
p1 = (short*) malloc(sizeof(short));
p2 = (int*) malloc(sizeof(int));
p3 = (float*) malloc(sizeof(float));
```

Allocating memory to pointer variable

```c
*p1=256;
*p2=100;
*p3=123.456;
```

Assigning values

```c
printf(“p1 is %hd\n”, *p1);
printf(“p2 is %d\n”, *p2);
printf(“p3 is %f\n”, *p3)
```

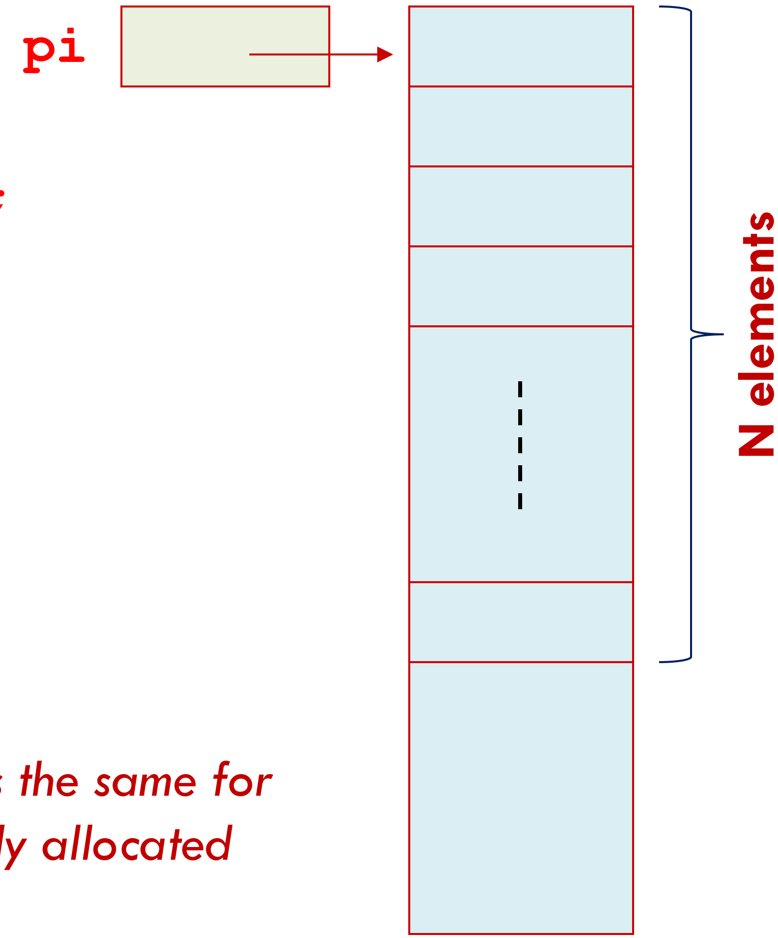
```
*p1 = 256
*p2 = 100
*p3 = 123.456001
```

Dynamically Allocated Arrays

Dynamically Allocated Arrays

Given

```
int *pi;
pi = (int *) malloc(N*sizeof(int));
```



Where does storage get allocated?

Not inside frames i.e., not inside call stack

But inside the heap

NOTE:

✓ *Usage syntax for the array and its elements is the same for static arrays (arrays on stack) and dynamically allocated arrays.*

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()    {
   int *p,*q,i;

    p = (int*) malloc(5*sizeof(int));

    q = p; // assigning array p to q

    for (i=0;i<5;i++)
      *p++ = i*i;

    for(i=0;i<5;i++)
      printf("Element at index %d is %d\n",i,q[i]);

    // for(i=0;i<5;i++)
    //  printf("Element at index %d is %d\n",i,*(q + i));
}
```

- This is not copying of array p into q.
- This means that the address contained in p is copied into q.
- Which means that now q also points to first location of the array p.

Both are equivalent

Dynamically Allocated Arrays

- We saw that arrays created inside a function could not be returned.
  - Since, arrays declared within a function are allocated in a frame and they are gone when function returns.
- Solutions:
  - Declare array outside function (in some other function) and pass as parameter (only starting address is passed)
    - *Passed by reference, so changes get reflected in the calling function.*
  - Or **declare a pointer** inside the function, **allocate memory** and **return** the pointer.
    - *Since, memory is allocated in the heap, it will remain in the calling function as well.*

Example: copy arrays - 1

The following program calls a function **copy()**, that copies the contents of one array into another.

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

void copy(int c[], int n, int d[])
{
    for (int i=0; i<n; i++)
    {
        d[i]=c[i];
        // or *(b+i)=*(a+i);
    }
}
```

```
int main()
{
    int a[5] = {1,2,3,4,5};
    int b[5];

    copy(a,5,b);

    for(int i=0;i<5;i++)
    {
        printf("%d\t",b[i]);
    }

}
```

Example: copy arrays – 2 (using pointers)

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

int * copy(int a[], int n)
{
    int * b = (int*) malloc(sizeof(int)*n);
    for (int i=0; i<n; i++)
    {
        b[i]=a[i];
        // or *(b+i)=*(a+i);
    }
    return b;
}
```

Can be replaced with
`int * a`

```
int main(){
    int a[5] = {1,2,3,4,5};
    int * b;

    b = copy(a,5);

    for(int i=0;i<5;i++){
        printf("%d\t",b[i]);
    }
}
```

Example: copy arrays – that is incorrect (using pointers)

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

void copy(int * a, int n, int * c)
{
    c = (int*) malloc(sizeof(int)*n);
    for (int i=0; i<n; i++)
    {
        c[i]=a[i];
    }
}
```

Will lead to segmentation fault (run-time error). Why?

The contents of **b** were copied into **c** during function call. **b** was empty (or garbage value) when it was passed. The memory that was allocated in **copy()** function, its first element's address is copied to **c** during the **malloc()** call. But this was not copied to **b**, when **copy()** returned.

Solution:

- *Allocate memory to* **b** *in* **main()** *and then pass. Try This!*
- *Allocate memory inside* **copy()** *and return. (previous slide)*
- *Use double pointer! (we will see later)*

```
int main(){
    int a[5] = {1,2,3,4,5};
    int * b;

    copy(a,5,b);

    for(int i=0;i<5;i++){
        printf("%d\t",b[i]);
    }
}
```

calloc()

- `calloc()` is used specifically for arrays and structures.
- It is more intuitive than `malloc()`.
- Example:

```
int *p = calloc (5, sizeof(int));
// 2 arguments in contrast to malloc()
```

- *The bytes created by* `calloc()` *are initialized to 0.*

Note:

- After `malloc()`, the block of memory allocated contains garbage values.
- `malloc()` is faster than `calloc()`

realloc()

What if the allocated number of bytes run out?

- We can reallocate!
- Use "realloc" from stdlib

```
int * pt = (int *)
      realloc(pt,2*MAX_SIZE*sizeof(int));
```

`realloc()` tries to

- Enlarge existing block
- Else, it creates a new block elsewhere and move existing data to the new block. It returns the pointer to the newly created block and frees the old block.
- If neither could be done, it returns a **`NULL`**.

Example

```c
int *p, *q, *r, i;
p = malloc(5*sizeof(int));
q = p;

for (i=0;i<5;i++)
  *p++ = i;
p = q;

printf("Array address %p \n", p);
r = realloc(p, 10*sizeof(int));
printf("Relocated address %p\n", r);
for(i=0; i<5; i++)
printf("The elements are %d\n", r[i]);
```

```
Array address 0xa27010

Relocated address 0xa27010

The elements are 0
The elements are 1
The elements are 2
The elements are 3
The elements are 4
```

Example

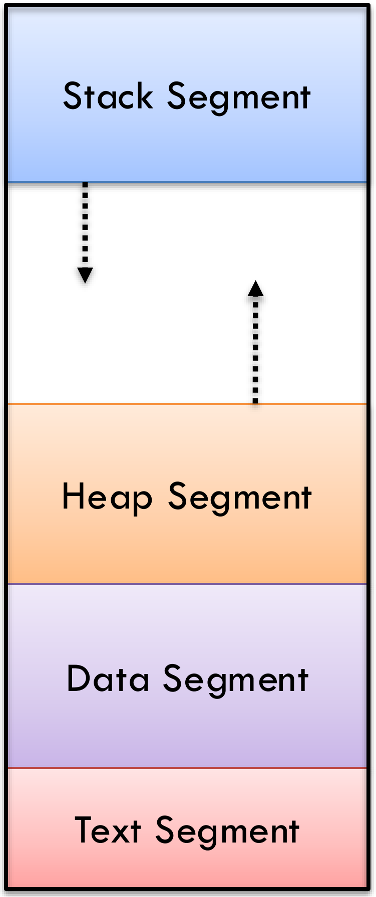
```
r = realloc(p, 10*sizeof(int));
```

⇓

```
r = realloc(p, 9000000*sizeof(int));
```

```
Array address 0x21c5010
Relocated address 0x7f72f15bc010
```

free()



- Block of memory freed using `free()` is returned to the heap
- Failure to free memory results in heap depletion.
  - which means that `malloc()` or `calloc()` can return `NULL` saying that no more memory is left to be allocated.
- `free()` syntax:

```
int * p = malloc(5*sizeof(int));
...
free(p);
```

Releases all `5*sizeof(int)` bytes allocated

innovate achieve lead

BITS Pilani
Pilani Campus

Memory Management Issues

Memory Management Issues

Dangling Pointer:

```
Address pointed by p = 0x8ff010
Address pointed by p = 0x8ff010
Address pointed by p = (nil)
```

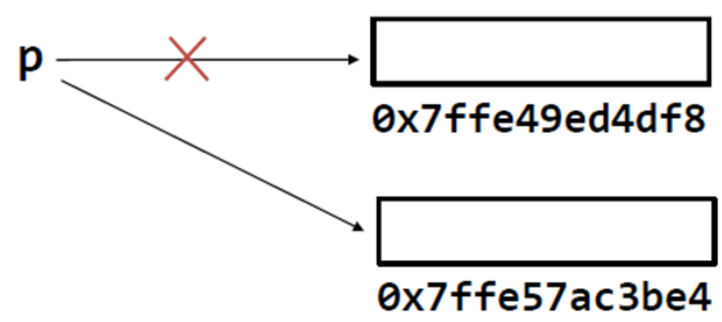
```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int * p, *q, *r,  i;
   p = (int*) malloc(sizeof(int));
   printf(“Address pointed by p = %p \n”, p);
   free(p); // p becomes dangling pointer
   printf(“Address pointed by p = %p \n”, p);
   p = NULL;
   printf(“Address pointed by p = %p \n”, p);
}
```

Memory Management Issues

Memory Leaks:

- **p** is made to point to another memory block without freeing the previous one.
- `1000*sizeof(int)` bytes previously allocated to **p** are now inaccessible.
- **This is a memory leak.** Leaked memory can be recovered only after the program terminates.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int * p, *q, *r,  i;
   p = (int*) malloc(1000*sizeo
   q = (int*) malloc(sizeof(int));
   p = q;
}
```



Another example of a memory leak

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int * p, *q, *r,  i;
   for (i=0;i<1000000;i++)
   {
      p = (int*) malloc(10000*sizeof(int));
     q = (int*) malloc(sizeof(int));
   }
   // some more lines of code
   ...
}
```

40,000 bytes are allocated and wasted for each iteration of this loop (considering sizeof(int) is 4 bytes)

Memory Leak (Consequences)

- *Memory leaks reduces the performance of the computer by reducing the amount of available memory.*
- *In the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down vastly.*
- *Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.*

Memory Management Practices

In C programming it is the responsibility of the programmer to:

- keep freeing the allocated memory after its usage is over so that `malloc()` and `calloc()` don't fail.
- prevent memory leaks by careful programming practices
- keep track of dangling pointers and re-use them.

Dynamically Allocated Array of Structures

Dynamically allocated struct variable and array of structures

```
struct stud {
    int roll;
    char dept_code[25];
    float cgpa;
} class, *ptr;
```

Rest of the operations are the same as structures and array of structures.
Only difference is that we have dynamically allocated memory in the heap.

```
struct stud * s1 = (struct stud*) malloc(sizeof
    (struct stud));
```

```
struct stud * studentArray = (struct stud*)
   malloc(sizeof(struct stud)*100);
```

Grocery Store Case

Grocery Store Case

You have to maintain a set of grocery items. Each grocery item has the following attributes: **ID** (Integer), **Name** (Char array), **Price** (float), **Quantity** (Integer). Implement the following functions:

- **readGroceryList()**: receives a **count** of grocery items as a function parameter, and reads the details of that many grocery items from the user and stores them in a dynamically allocated array, and returns it
- **printGroceryList()**: that receives an **array of grocery items** and its **count** as parameters, and prints the details of all the grocery items present in that array.
- **findItem()**: searches for the first grocery item in an **array of grocery items**, whose quantity is equal to **qVal** (function parameter) and returns that item
- **findMaxPriceItem()**: searches for the grocery item in an **array of grocery items** that has maximum price and returns it

Grocery Store Case: Structure Definition

```
struct item
{
    int ID;
    char name[50];
    float price;
    int quantity;
};
```

Grocery Store Case: readGroceryItems()

```
struct item * readGroceryList(int count){
    struct item * gItems = (struct item *) malloc(sizeof(struct
   item)*count);

    int uniqNum = 1;

    for (int i = 0; i < count; i++)
    {
        gItems[i].ID = uniqNum++;
        printf("\nEnter details for item %d:\n",i+1);
        printf("Name:");
        scanf("%s",gItems[i].name);
        printf("Price:");
        scanf("%f",&gItems[i].price);
        printf("Quantity:");
        scanf("%d",&gItems[i].quantity);
    }

    return gItems;
}
```

Grocery Store Case: printGroceryList()

```
void printGroceryList(struct item * gItems, int count)
{
    for (int i = 0; i < count; i++)
    {
        printf("Item ID: %d, ", gItems[i].ID);
        printf("Name: %s, ", gItems[i].name);
        printf("Price: %f, ", gItems[i].price);
        printf("Quantity: %d\n", gItems[i].quantity);
    }
}
```

Grocery Store Case: findItem()

```
struct item findItem(int qVal, struct item * gItems, int count)
{
    int i = 0; int index = -1;
    while(i<count)
    {
        if(gItems[i].quantity == qVal)
        {
            index = i;
            return gItems[index];
        }
        i++;
    }

    struct item emptyItem;
    emptyItem.ID = -1;
    return emptyItem;
}
```

Grocery Store Case: findMaxPriceItem()

```
struct item findMaxPriceItem(struct item * gItems, int count)
{
   int maxIndex = -1;
   int maxPrice = -1;
   int i = 0;

   while(i<count)
   {
        if(gItems[i].price > maxPrice)
        {
             maxPrice = gItems[i].price;
             maxIndex = i;
        }
        i++;
   }

   return gItems[maxIndex];
}
```

Grocery Store Case: main()

```
int main(){
    int num_g;
    printf("Enter number of unique grocery items in the store:");
    scanf("%d",&num_g);
    struct item * gItems = readGroceryList(num_g);

    printGroceryList(gItems,num_g);

    ... // contd. next page

}
```

Grocery Store Case: main() (contd.)

```
int main(){
    ... // contd. from previous page

    int qVal;
    printf("\nEnter the quantity of the item you wish to find: ");
    scanf("%d",&qVal);
    struct item fItem= findItem(qVal,gItems,num_g);

    if(fItem.ID == -1)   printf("\nItem Not Found!\n");
    else
    {
      printf("Item with quantity %d is %s\n", qVal, fItem.name);
    }

    struct item maxItem = findMaxPriceItem(gItems,num_g);
    printf("The item with maximum price is %s\n", maxItem.name);

}
```

Sample Execution

Enter number of unique grocery items in the store:**3**

Enter details for item 1:

Name: **Hamam**

Price: **20.5**

Quantity: **15**

Enter details for item 2:

Name: **Sugar**

Price: **90.25**

Quantity: **20**

Enter details for item 3:

Name: **Milkmaid**

Price: **210.65**

Quantity: **6**

Item ID: 1, Name: Hamam, Price: 20.500000, Quantity: 15

Item ID: 2, Name: Sugar, Price: 90.250000, Quantity: 20

Item ID: 3, Name: Milkmaid, Price: 210.649994, Quantity: 6

Enter the quantity of the item you wish to find: **20**

Item with quantity 20 is Sugar

The item with maximum price is Milkmaid

innovate achieve lead

BITS Pilani
Pilani Campus

Multi-dimensional Arrays using dynamic memory allocation

Multi-dimensional arrays

2D arrays in stack memory:

```
#define NUM_ROWS 3
#define NUM_COLS 2
int arr2D[NUM_ROWS][NUM_COLS];
```

Notice the double pointer

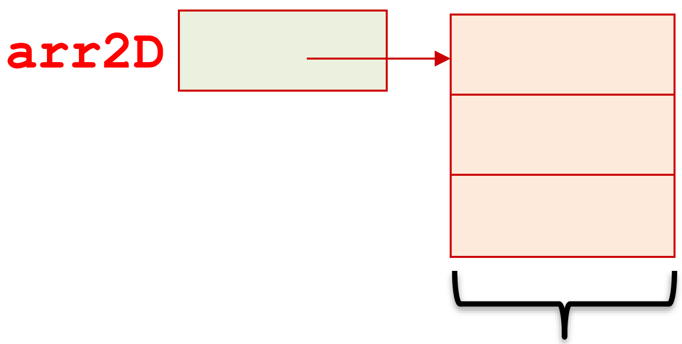
This array of 3 rows and 2 columns resides in stack and has all its problems that we discussed earlier for 1D arrays

We can declare this array in heap dynamically at run-time by using a pointer to a groups of pointers:

```
int ** arr2D = (int **) malloc(NUM_ROWS*sizeof(int*));
for (int i=0; i<NUM_ROWS; i++)
{
   arr2D[i] = (int*) malloc(NUM_COLS*sizeof(int));
}
arr2D[1][2] = 10; // is a valid assignment
```

2D Array

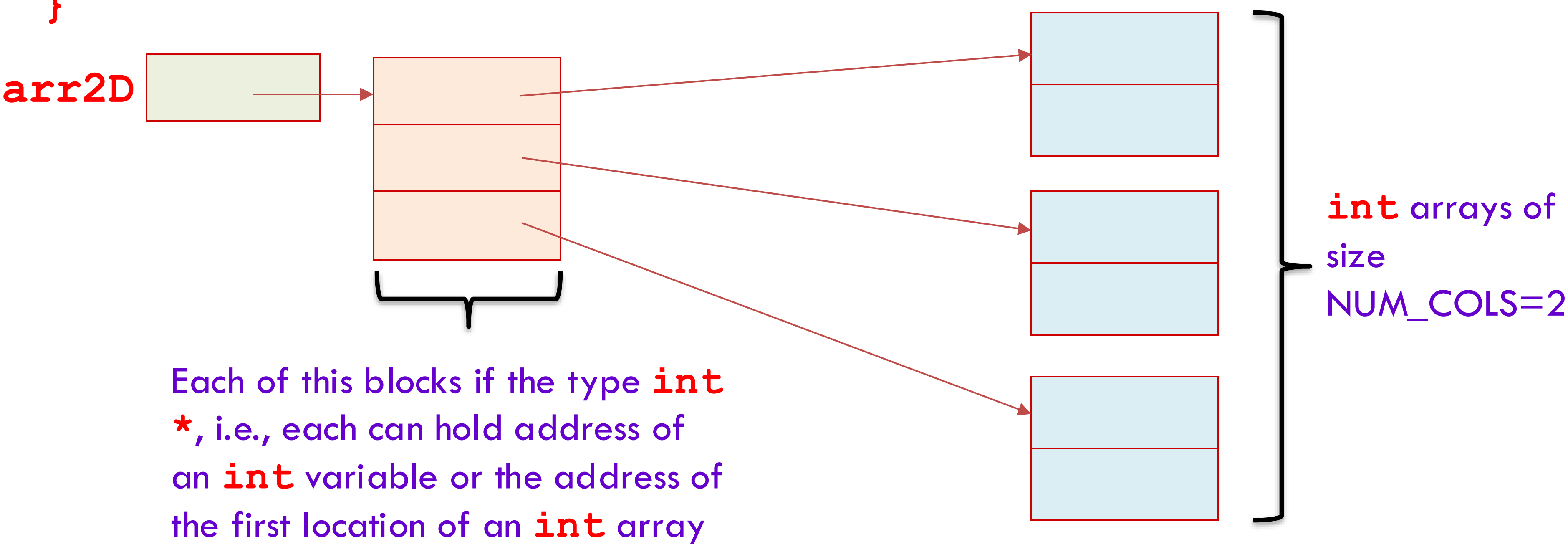
```
#define NUM_ROWS 3
#define NUM_COLS 2
int ** arr2D = (int **) malloc(NUM_ROWS*sizeof(int*));
...
```



Each of this blocks if the type `int *`, i.e., each can hold address of an `int` variable or the address of the first location of an `int` array

2D Array (contd.)

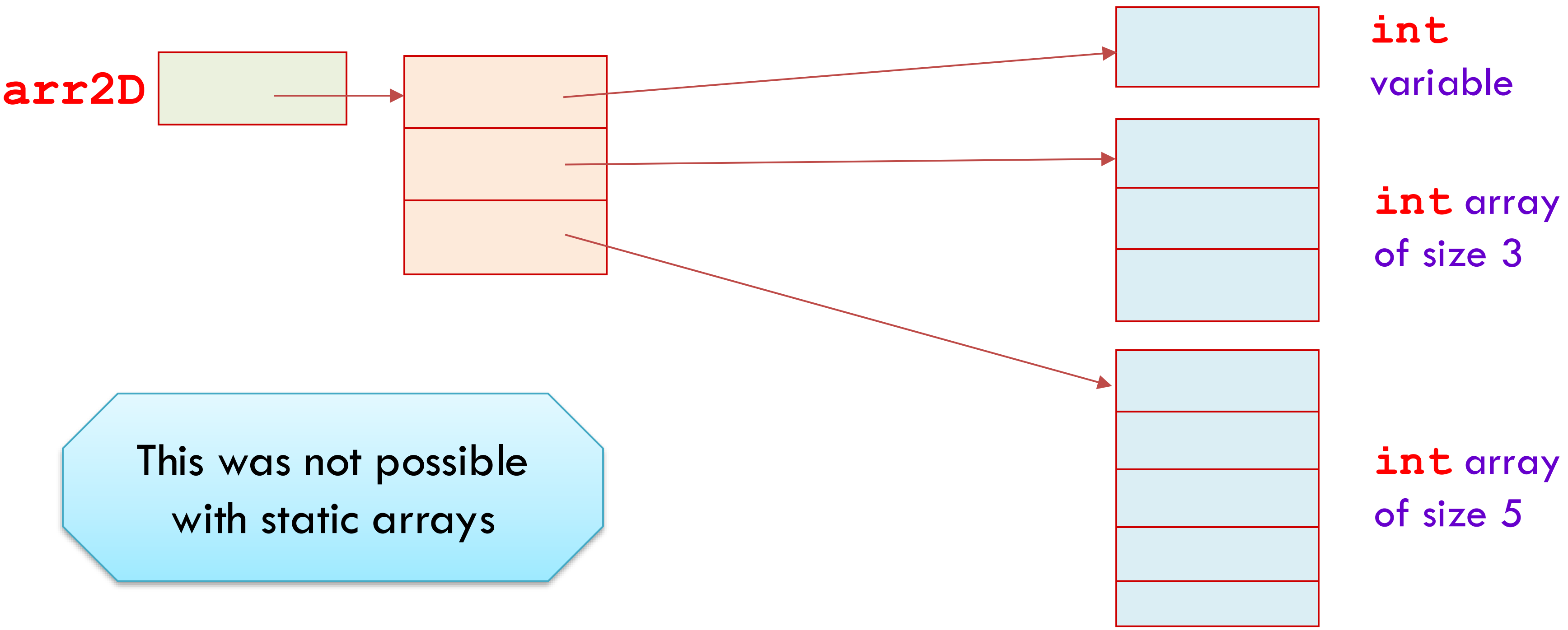
```
…
for (int i=0; i<NUM_ROWS; i++)
{
   arr2D[i] = (int*) malloc(NUM_COLS*sizeof(int));
}
```



2D Array (variable size)

One can define each array of variable size as well

```
arr2D[0] = (int*) malloc(sizeof(int)); // single int variable
arr2D[0] = (int*) malloc(sizeof(int)*3); // int array of size 3
arr2D[0] = (int*) malloc(sizeof(int)*5); // int array of size 5
```



Static vs Dynamic Arrays

Arrays (static) are Pointers in C?

- Well, not exactly …
- Arrays cannot be reallocated (starting address and size are fixed).
- Multi-dimensional arrays (static) are not pointers to pointers.
  - They are contiguous memory locations

```
int a[3][3];
```

```
  a          a+1        a+2        a+3        a+4
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1],
  a+5        a+6        a+7        a+8
a[1][2], a[2][0], a[2][1], a[2][2]
```

Exercises

- Write a C program that computes the transpose of a 2D array.
- Write a C program that receives a 2D array and sorts the elements by accessing the 2D array as a 1D array.

Remember our incorrect copy arrays function…


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

void copy(int * a, int n, int * c)
{
     c = (int*) malloc(sizeof(int)*n);
     for (int i=0; i<n; i++)
     {
          c[i]=a[i];
     }
}
```

Will lead to segmentation fault (run-time error). Why?

The contents of b were copied into c during function call. b was empty (or garbage value) when it was passed. The memory that was allocated in copy() function, its first element's address is copied to c during the malloc() call. But this was not copied to b, when copy() returned.

```
int main(){
     int a[5] = {1,2,3,4,5};
     int * b;

     copy(a,5,b);

     for(int i=0;i<5;i++){
          printf("%d\t",b[i]);
     }
}
```

Solution:

- *Allocate memory to b in main() and then pass. Try This!*
- *Allocate memory inside copy() and return. (last slide)*
- ***Use double pointer! (we will see NOW)***

Example: copy arrays – that is incorrect (using pointers)

```
#include <stdio.h>
#include <stdlib.h>
void copy(int c[], int n, int ** d) {
    *d = (int *) malloc(n*sizeof(int));
    for (int i=0; i<n; i++)
    {
        (*d)[i]=c[i];
    }
}
```

Use of double pointer enables us to not return anything, yet change getting reflected in main(). This is call by reference for dynamically allocated arrays

```
int main(){
    int a[5] = {1,2,3,4,5};
    int ** b =  (int **) malloc(sizeof(int *));

    copy(a,5,b);

    for(int i=0;i<5;i++){
        printf("%d\t",(*b)[i]);
    }
}
```

innovate achieve lead

BITS Pilani
Pilani Campus

Command Line Arguments

Command Line Arguments

```
#include <stdio.h>
/* echo */
int main(int argc, char **argv)
{
// argc - number of arguments passed
// argv[0] is the name of the
(command) file being executed
for (j=1; j < argc; j++)
   printf("%s ", argv[j]);
return 0;

}
```

- `gcc echo.c -o echoTest`
- `./echoTest How do you do?`
- Output: **How do you do?**

- Observe that we printing argv[j] from j=1
  - argv[0] refers to the name of the executable (in this case "echoTest").
  - Each other argv[j] refers to one word (separated by blank spaces) of command line argument.
  - argc has the count of arguments

Example 1

```
int main(int argc, char *argv[])
{
  printf(“Number of arguments: %d \n", argc);
  int i = 0;
  while(i < argc){
    printf("Argument %d: %s \n",i, argv[i]);
    i++;
  }
  return 0;
}
```

Notice the change

```
amitesh@Prithvi:~$ gcc test1.c
amitesh@Prithvi:~$ ./a.out hello world
```

```
Number of arguments is 3
```

```
Argument 0: ./a.out
Argument 1: hello
Argument 2: world
```

Example 2

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int a, b, c;

    a = atoi(argv[1]);
    b = atoi(argv[2]);
    c = atoi(argv[3]);

    printf("\n %d",a+b+c);
    return 0;
}
```

innovate achieve lead

BITS Pilani
Pilani Campus

Thank you

Q & A