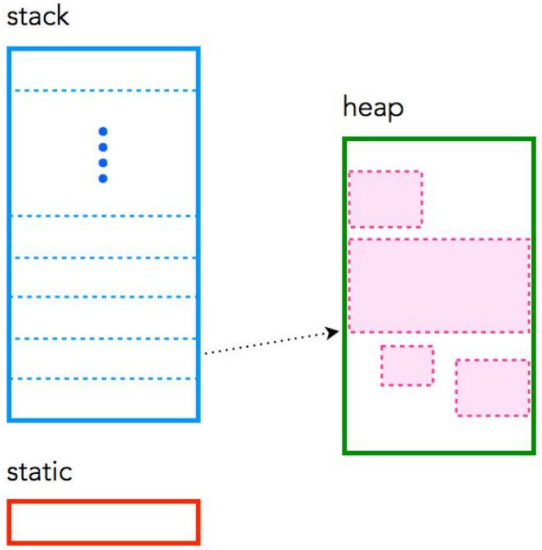
אוניברסיטת
אריאל
בשומרון

Memory Management

Memory Organization

- C has three different pools of memory -
  - **stack**: local variable storage (automatic, continuous memory).
  - **Static allocation**: global variable storage, permanent for the entire run of the program.
  - **Dynamic heap**: dynamic global storage (large pool of memory, not allocated in continuous order).



Stack

אוניברסיטת אריאל בשומרון

- Maintains memory during function calls:
  - Argument of the function
  - Local variables
- Limited “lifetime” - Variables on the stack have limited “life time”
- Constant size - stack size must be defined at compile time

Stack

אוניברסיטת אריאל בשומרון

- *Stack* is used to store variables used inside a function, including the main() function.
- Scope - this leads to the local scope of function variables.
- Order - it's a LIFO structure, "**L**ast-**I**n,-**F**irst-**O**ut".
- New variable - every time a function declares a new variable it is "pushed" onto the stack.
- End of Process - when a function finishes running, all the variables associated with that function on the stack are deleted, and the memory they use is freed up.
- Memory allocation - stack is a special region of memory, and automatically managed by the CPU – so you don't have to allocate or deallocate memory.

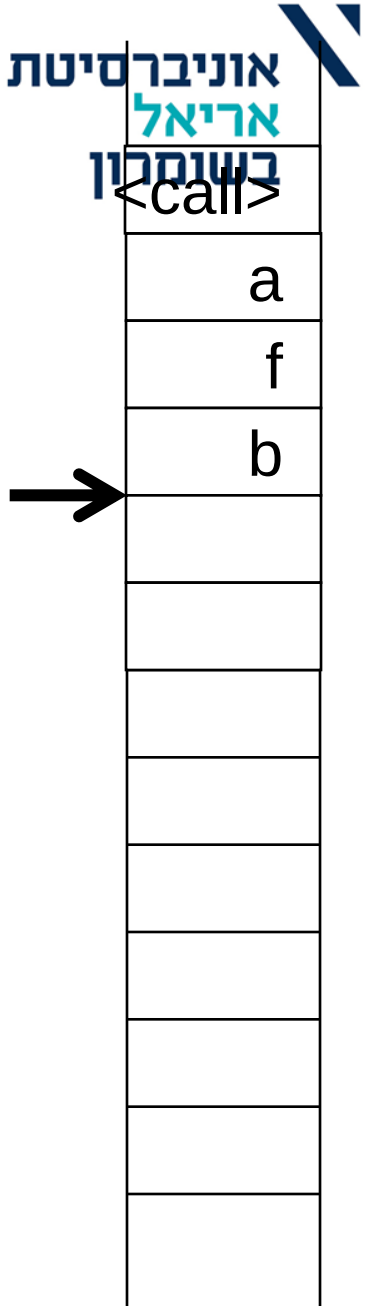
Stack

אוניברסיטת אריאל בשומרון

- Stack memory - is divided into successive frames where each time a function is called, it allocates itself a fresh stack frame.
- Stack size - there is generally a limit on the size of the stack – which can vary with the operating system.
- Stack overflow - if a program tries to put too much information on the stack, stack overflow will occur. Stack overflow happens when all the memory in the stack has been allocated, and further allocations begin overflowing into other sections of memory.
- Stack overflow also occurs in situations when recursion is incorrectly used.

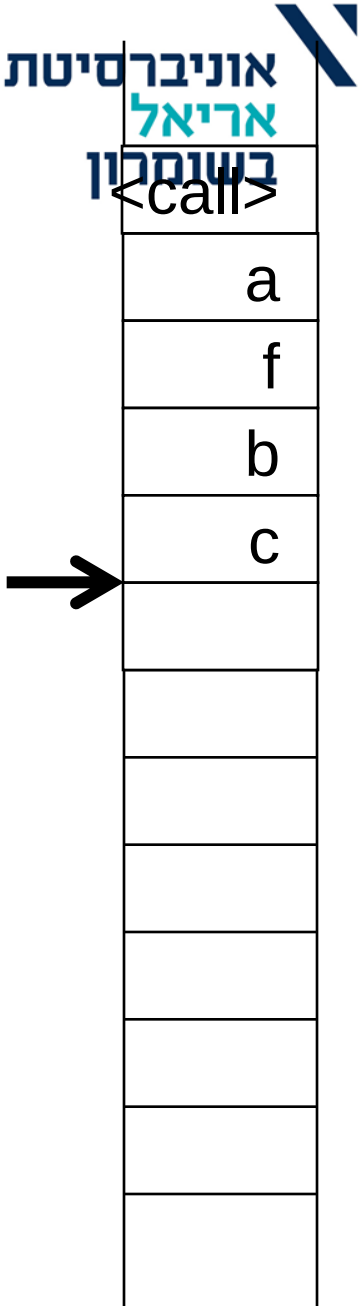
Stack – memory allocation

```
int foo( int a, double f )
{
  int b;
  ...
  {
    int c;
    ...
  }
  ...
}
```



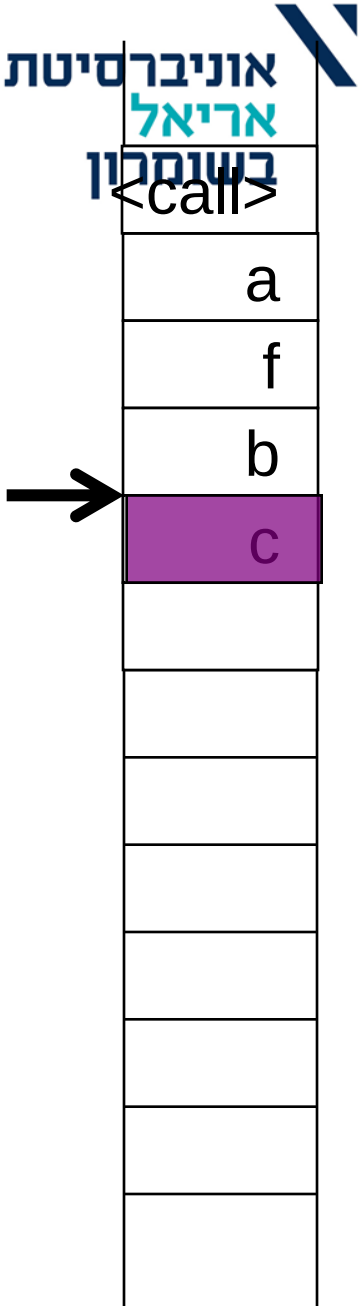
Stack – memory allocation

```
int foo( int a, double f )
{
    int b;
    ...
    {
        int c;
        ...
    }
    ...
}
```



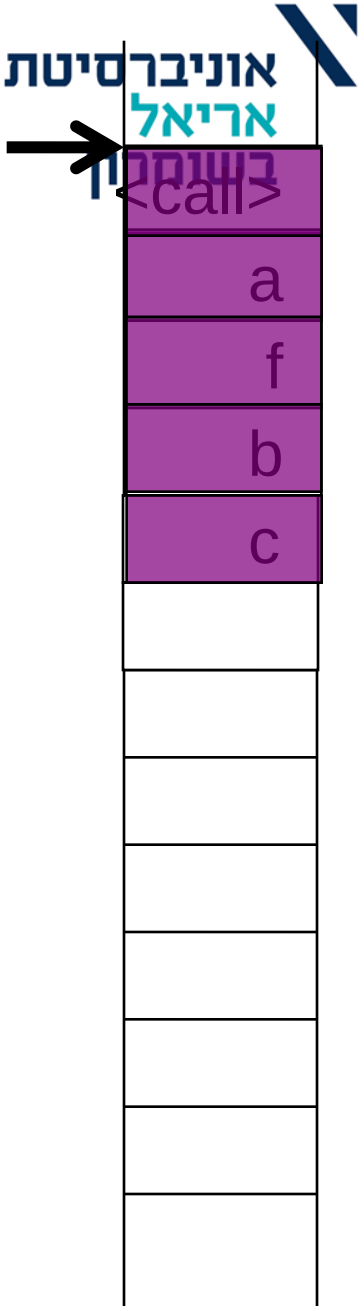
Stack – memory allocation

```
int foo( int a, double f )
{
  int b;
  ...
  {
    int c;
    ...
  }
  ...
}
```



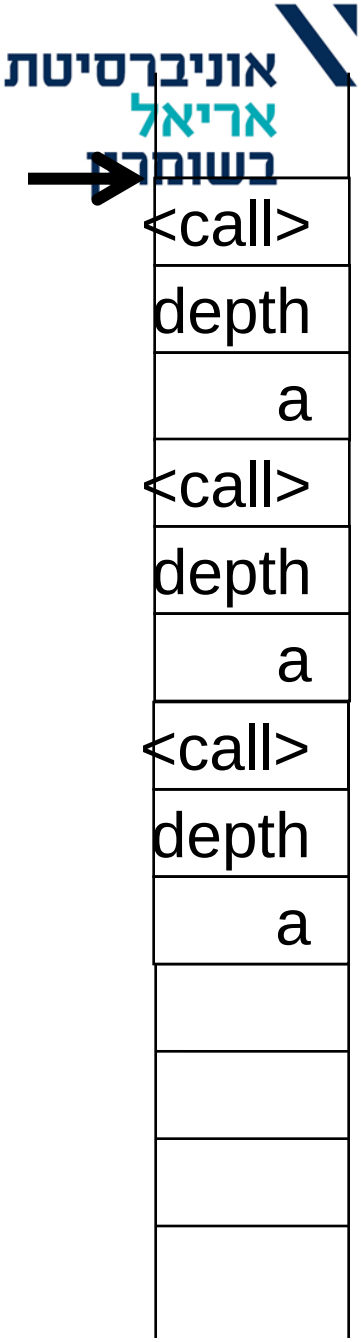
Stack – memory allocation

```
int foo( int a, double f )
{
  int b;
  ...
  {
    int c;
    ...
  }
  ...
}
```



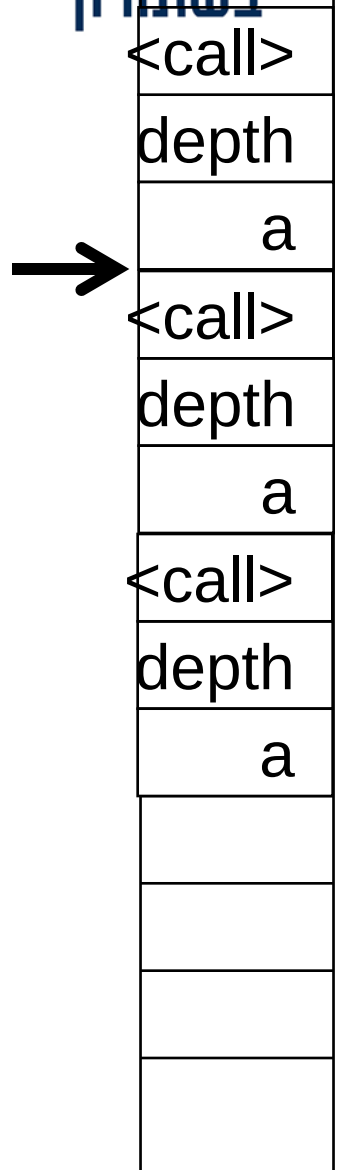
Stack – memory allocation

```
void foo( int depth )
{
  int a;
  if( depth > 1 )
  {
     foo( depth-1 );
  }
}
int main()
{
  foo(3);
  …
```



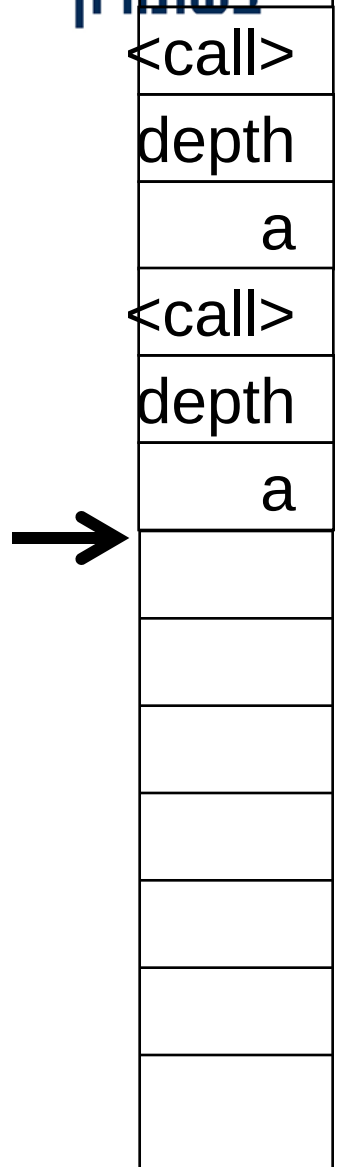
Stack – memory allocation

```
void foo( int depth )
{
  int a;
  if( depth > 1 )
  {
     foo( depth-1 );
  }
}
int main()
{
  foo(3);
  …
```



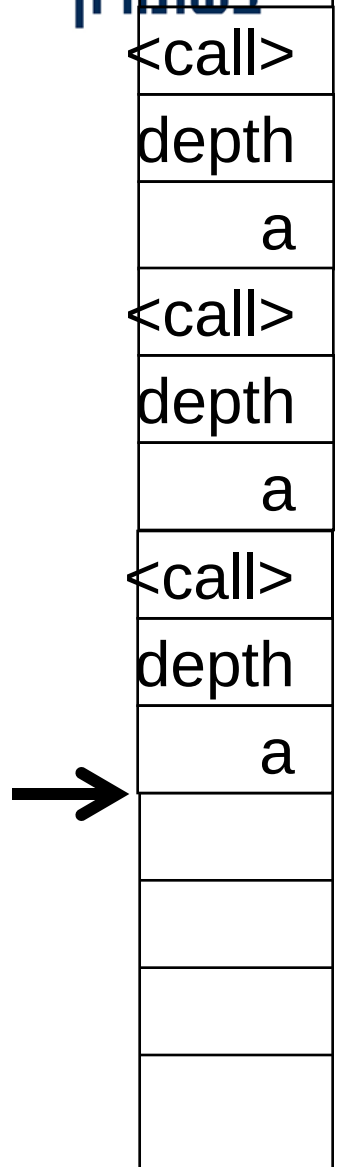
Stack – memory allocation

```
void foo( int depth )
{
  int a;
  if( depth > 1 )
  {
     foo( depth-1 );
  }
}
int main()
{
  foo(3);
  …
```



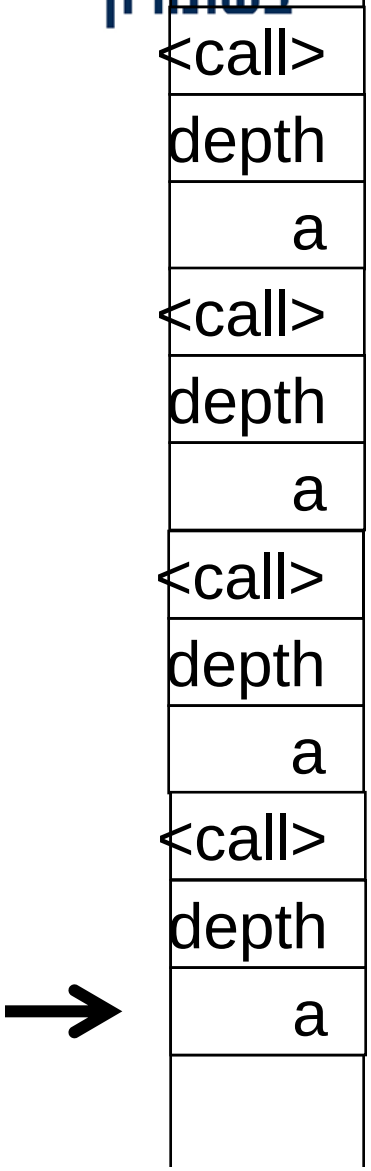
Stack – memory allocation

```
void foo( int depth )
{
  int a;
  if( depth > 1 )
  {
     foo( depth-1 );
  }
}
int main()
{
  foo(3);
  …
```



Stack – memory allocation

```
void foo( int depth )
{
  int a;
  if( depth > 1 )
  {
     foo( depth-1 );
  }
}
int main()
{
  foo(3);
  …
```



Stack – overflow error

אוניברסיטת אריאל בשומרון

```
void foo( int depth )
{
  int a;
  if( depth > 1 )
  {
    foo( depth );
  }
}
```

Iteration can result in run time error:

out of stack space

Stack – summary

אוניברסיטת אריאל בשומרון

- Stack is managed by the CPU, there is no ability to modify it
- Variables are allocated and freed automatically
- The stack has an upper bound of size
- The stack grows and shrinks as variables are created and destroyed
- Stack variables only exist whilst the function that created them exists

Global memory – *Static* allocation

אוניברסיטת אריאל בשומרון

- Memory allocation – static memory for global variables – must be defined in compile time
- Memory "lifetime" - static memory persists throughout the entire life of the whole program (not the specific function) , and is usually used to store things like *global* variables, or variables created with the **static** clause.
- Global memory can come from one of two places –
  - If a variable is declared *outside* of a function, it is considered global, meaning it is accessible anywhere in the program. Global variables are static, and there is only one copy for the entire program. Inside a function the variable is allocated on the stack.
  - It is also possible to force a variable to be static using the static clause -

```
static int variable1
```

Static allocation - limitations

אוניברסיטת אריאל בשומרון

- Static global allocation is ill-posed because it requires predefined memory size -
  - Too small - the program cannot handle sizes larger than were specified
  - Too large - if we allocate very large memory, then it may be a waste if we will not use all of it

Therefore, we want to use memory on "as needed" basis

Dynamic allocation - heap

אוניברסיטת אריאל בשומרון

- Dynamic global memory - can be allocated and freed during run time.
- Allocation control - the compiler (not the user) controls how much is allocated and when.
- The *heap* is the opposite of the stack. The *heap* is a large pool of memory that can be used dynamically – it is also known as the “free store”.
- Memory management function – this memory is not automatically managed – you have to explicitly allocate (using functions such as malloc), and deallocate (e.g. free) the memory.
- Memory Leak - Failure to free the memory when you are finished with it will result in what is known as a *memory leak* – memory that is still “being used”, and not available to other processes.
- Memory size - unlike the stack, there are generally no restrictions on the size of the heap, other than the available memory in the machine. Variables created on the heap are accessible anywhere in the program.

Dynamic allocation - why

אוניברסיטת אריאל בשומרון

- Size allocation - static array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

- Sometimes the size of static array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time.

- To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used. These functions are defined in the <stdlib.h> header file

Dynamic allocation – heap - Summary

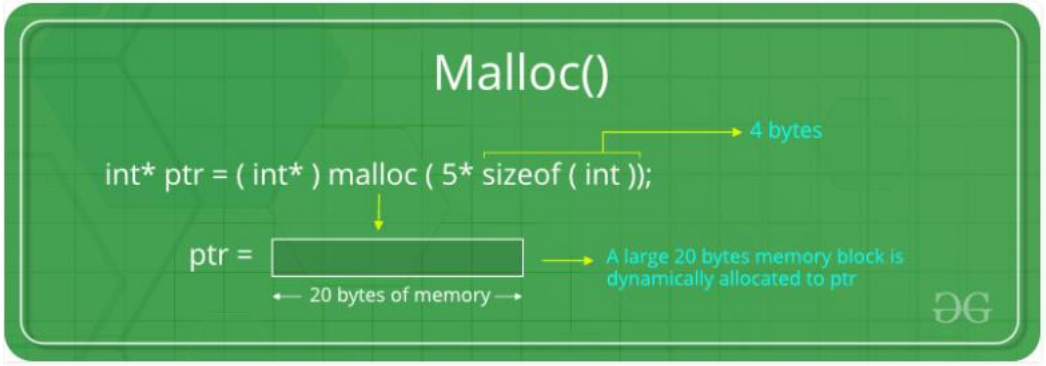
- Heap memory requires using pointers to access it.
- Heap is managed by the programmer
- The ability to modify heap is somewhat boundless
- Variables are allocated and freed using functions like malloc() and free()
- The heap is large, and is usually limited by the available memory

malloc – memory allocation

- The name "malloc" stands for memory allocation
- malloc() function reserves a block of memory of specified number of bytes.
- If space is insufficient, allocation fails and returns a NULL pointer.
- malloc syntax –

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

- Example -



malloc - Dynamic Memory allocation

אוניברסיטת אריאל בשומרון

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

Malloc - Dynamic Memory allocation

The variable **x** is static storage, because of its global nature. **str** is a dynamic stack storage which is deallocated when the program ends. The function **malloc()** is used to allocate 100 pieces of dynamic heap storage, each the size of char, to **str**. Conversely, the function **free()**, deallocates the memory associated with **str**.

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

int x;

int main(void)
{
    int y;
    char *str;

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```

Malloc – success checking

אוניברסיטת אריאל בשומרון

- Always check allocation success

```
char *str = (char*)malloc(5*sizeof(char));
if (str==NULL)
{
  //print error message
        exit(1); //or perform relevant operation
}
```

calloc – memory allocation

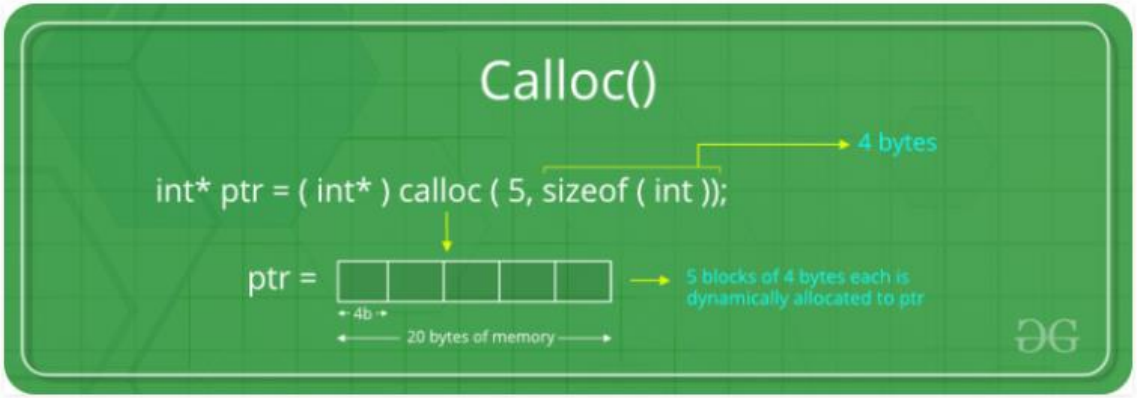
אוניברסיטת אריאל בשומרון

- The name "*calloc*" stands for contiguous allocation.
- The malloc() function allocates memory and leaves the memory uninitialized.

  Whereas, the calloc() function allocates memory and initializes all bits to '0'.
- Syntax of *calloc*

```
ptr = (castType*)calloc(n, size);
```

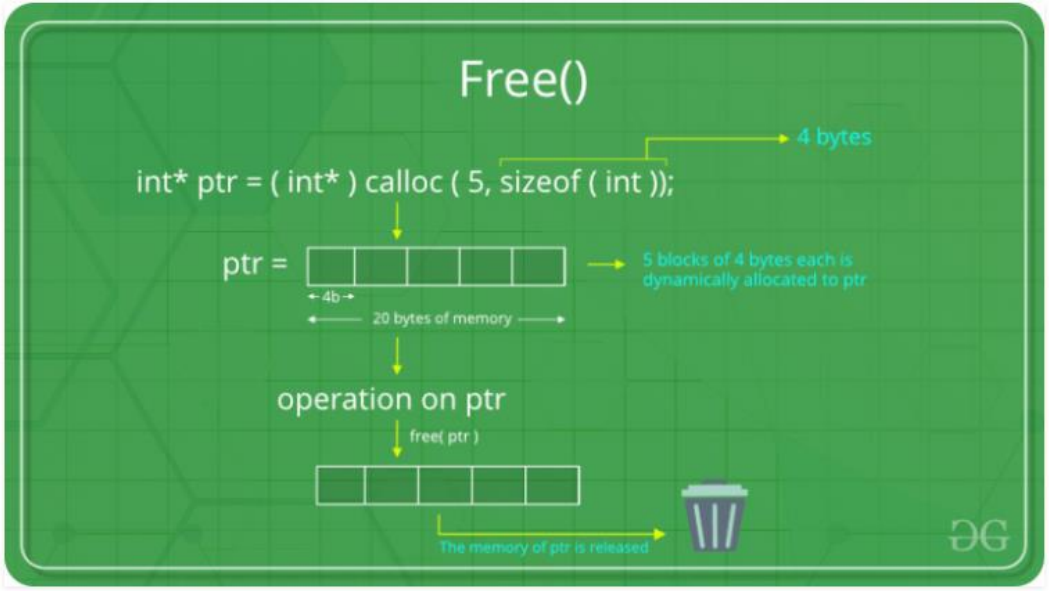
- Example –



free – deallocating memory

- **“free”** method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated automatically on their own. Hence the free() method is used, to explicitly return the memory block pointed by p to the pool of unused memory. It helps to reduce wastage of memory by freeing it.
- General syntax -

```
free(ptr);
```



free & malloc

אוניברסיטת אריאל בשומרון

```c
// Program to calculate the sum of n numbers entered by the user

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

int main()
{
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int));

    // if memory cannot be allocated
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);

    // deallocating the memory
    free(ptr);

    return 0;
}
```

realloc ()

- If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

- General syntax

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

Here, ptr is reallocated with a new size x

Example - realloc ()


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

int main()
{
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Addresses of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
         printf("%u\n",ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // rellocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory: ");
    for(i = 0; i < n2; ++i)
         printf("%u\n", ptr + i);

    free(ptr);

    return 0;
}
```

Dynamic memory allocation for structure

אוניברסיטת אריאל בשומרון

```c
#include <stdio.h>
#include <stdlib.h>
struct course {
    int marks;
    char subject[30];
};

int main() {
    struct course *ptr;
    int i, noOfRecords;
    printf("Enter the number of records: ");
    scanf("%d", &noOfRecords);

    // Memory allocation for noOfRecords structures
    ptr = (struct course *)malloc(noOfRecords * sizeof(struct course));
    for (i = 0; i < noOfRecords; ++i) {
        printf("Enter the name of the subject and marks respectively:\n");
        scanf("%s %d", (ptr + i)->subject, &(ptr + i)->marks);
    }

    printf("Displaying Information:\n");
    for (i = 0; i < noOfRecords; ++i)
        printf("%s\t%d\n", (ptr + i)->subject, (ptr + i)->marks);

    return 0;
}
```

Dynamic structure allocation - NULL

אוניברסיטת אריאל בשומרון

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
 struct stock {
 char symbol[5];
 int quantity;
 float price;
 };
 struct stock *invest;
/* Create structure in memory */
 invest=(struct stock *)malloc(sizeof(struct stock));
 if(invest==NULL)
 {
 puts("Some kind of malloc() error");
 exit(1);
 }
/* Assign structure data */
 strcpy(invest->symbol,"GOOG");
 invest->quantity=100;
 invest->price=801.19;
/* Display database */
 puts("Investment Portfolio");
 printf("SymboltSharestPricetValuen");
 printf("%-6st%5dt%.2ft%.2fn",
 invest->symbol,
 invest->quantity,
 invest->price,
 invest->quantity*invest->price);
 return(0);
}
```

Bad destructors

```
struct Vec {
  int *_arr;
  size_t _length;
};

void freeVec (struct Vec *v) {
  free (v->_arr);
  free (v);
}

int main () {
  struct Vec *v = newVec(5);
  // do something
  freeVec (v);
}
```

But if v is NULL?

Good destructors

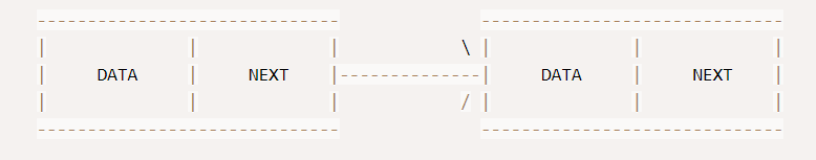
```
struct Vec {
  int *_arr;
  size_t _length;
};

void freeVec (struct Vec *v) {
  if (v==NULL) return;
  free (v->_arr);
  free (v);
}

int main () {
  struct Vec *v = newVec(5);
  // do something
  freeVec (v);
}
```

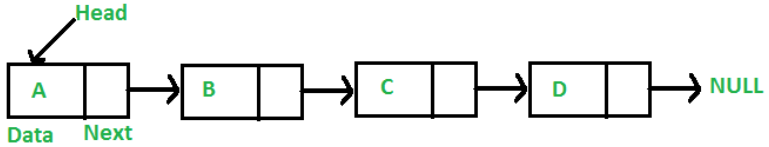
אוניברסיטת
אריאל
בשומרון

Linked Lists

Linked Lists

- Linked lists are the best and simplest example of a dynamic data structure that uses pointers for its implementation.
- Linked lists function as dynamic array that can grow and shrink as needed, from any point in the array.
- Linked list is a set of dynamically allocated nodes, arranged in such a way that each node contains one value and one pointer. The pointer always points to the next member of the list. If the pointer is NULL, then it is the last node in the list.





- A linked list is held using a local pointer variable which points to the first item of the list. If that pointer is also NULL, then the list is considered to be empty.

Linked Lists – pros and cons

אוניברסיטת אריאל בשומרון

- Linked lists have a few advantages over arrays:
  - **Ease of insertion/deletion** - items can be added or removed from the middle of the list. In arrays, inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.
  - **Dynamic size** - there is no need to define an initial size. The size of the arrays is fixed, so we must know the upper limit on the number of elements in advance.
- Linked lists also have a few disadvantages:
  - **Random access** is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
  - **Dynamic memory allocation** and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
  - **Linked lists have a much larger overhead over arrays**, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.

Linked Lists

- Let’s define a list node – our node type is *node_t*

```c
typedef struct node {
    int val;
    struct node * next;
} node_t;
```

- Now we can use the nodes. Let's create a local variable which points to the first item of the list (called *head*).

```c
node_t * head = NULL;
head = (node_t *) malloc(sizeof(node_t));
if (head == NULL) {
    return 1;
}

head->val = 1;
head->next = NULL;
```

- We've just created the first variable in the list. We must set the value, and the next item to be empty, if we want to finish populating the list. Notice that we should always check if *malloc* returned a NULL value or not.

Exercise

- Let’s create a linked list with 3 nodes

```
// A simple C program to introduce
// a linked list
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

// Program to create a simple linked
// list with 3 nodes
int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    /* Three blocks have been allocated dynamically.
     We have pointers to these three blocks as head,
     second and third
       head           second          third
        |               |               |
        |               |               |
    +---+-----+     +----+----+     +----+----+
    | #  | #  |     | #  | #  |     |  # |  # |
    +---+-----+     +----+----+     +----+----+

   # represents any random value.
   Data is random because we haven’t assigned
   anything yet  */

    head->data = 1; // assign data in first node
    head->next = second; // Link first node with
    // the second node

    /* data has been assigned to the data part of the first
     block (block pointed by the head). And next
     pointer of first block points to second.
     So they both are linked.
```

```
   head          second         third
    |             |              |
    |             |              |
  +---+---+     +----+----+     +-----+----+
  | 1  | o----->| #  | #  |     |  #  | #  |
  +---+---+     +----+----+     +-----+----+
*/

  // assign data to second node
  second->data = 2;

  // Link second node with the third node
  second->next = third;

  /* data has been assigned to the data part of the second
   block (block pointed by second). And next
   pointer of the second block points to the third
   block. So all three blocks are linked.

     head         second         third
      |            |              |
      |            |              |
  +---+---+     +---+---+     +----+----+
  | 1  | o----->| 2 | o-----> |  # |  # |
  +---+---+     +---+---+     +----+----+      */

  third->data = 3; // assign data to third node
  third->next = NULL;

  /* data has been assigned to data part of third
  block (block pointed by third). And next pointer
  of the third block is made NULL to indicate
  that the linked list is terminated here.

   We have the linked list ready.

         head
           |
           |
     +---+---+     +---+---+       +----+------+
     | 1  | o----->|  2  | o-----> |  3 | NULL |
     +---+---+     +---+---+       +----+------+

  Note that only head is sufficient to represent
  the whole list.  We can traverse the complete
  list by following next pointers.    */

  return 0;
}
```

Linked Lists

- To add a variable to the end of the list, we can just continue advancing to the next pointer –

```
node_t * head = NULL;
head = (node_t *) malloc(sizeof(node_t));
head->val = 1;
head->next = (node_t *) malloc(sizeof(node_t));
head->next->val = 2;
head->next->next = NULL;
```

- This can go on and on, but what we should actually do is advance to the last item of the list, until the next variable will be NULL.

Iterating over a list

- Let's build a function that prints out all the items of a list.
- To do this, we need to use a current pointer that will keep track of the node we are currently printing. After printing the value of the node, we set the current pointer to the next node, and print again, until we've reached the end of the list (the next node is NULL).

Iterating over a list

- Let's build a function that prints out all the items of a list.
- To do this, we need to use a current pointer that will keep track of the node we are currently printing. After printing the value of the node, we set the current pointer to the next node, and print again, until we've reached the end of the list (the next node is NULL).

```
void print_list(node_t * head) {
    node_t * current = head;

    while (current != NULL) {
        printf("%d\n", current->val);
        current = current->next;
    }
}
```

Adding an item to the end of the list

- We set the pointer to start from the head and then in each step, we advance the pointer to the next item in the list, until we reach the last item.

Adding an item to the end of the list

- We set the pointer to start from the head and then in each step, we advance the pointer to the next item in the list, until we reach the last item.

```
void push(node_t * head, int val) {
    node_t * current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    /* now we can add a new variable */
    current->next = (node_t *) malloc(sizeof(node_t));
    current->next->val = val;
    current->next->next = NULL;
}
```

Adding an item to the beginning of the list (pushing to the list)

אוניברסיטת אריאל בשומרון

- To add to the beginning of the list, we will need to do the following:
  - Create a new item and set its value
  - Link the new item to point to the head of the list
  - Set the head of the list to be our new item
- This will effectively create a new head to the list with a new value, and keep the rest of the list linked to it.
- We want to be able to modify the head variable. To do this, we must pass a pointer to the pointer variable (a double pointer) so we will be able to modify the pointer itself

Adding an item to the beginning of the list (pushing to the list)

אוניברסיטת אריאל בשומרון

- To add to the beginning of the list, we will need to do the following:
  - Create a new item and set its value
  - Link the new item to point to the head of the list
  - Set the head of the list to be our new item
- This will effectively create a new head to the list with a new value, and keep the rest of the list linked to it.
- We want to be able to modify the head variable. To do this, we must pass a pointer to the pointer variable (a double pointer) so we will be able to modify the pointer itself

```
void push(node_t ** head, int val) {
    node_t * new_node;
    new_node = (node_t *) malloc(sizeof(node_t));

    new_node->val = val;
    new_node->next = *head;
    *head = new_node;
}
```

Removing the first item (popping form the list)

אוניברסיטת אריאל בשומרון

- To pop a variable, we will need to reverse this action:
  - Take the next item that the head points to and save it
  - Free the head item
  - Set the head to be the next item that we've stored on the side

```
int pop(node_t ** head) {
    int retval = -1;
    node_t * next_node = NULL;

    if (*head == NULL) {
        return -1;
    }

    next_node = (*head)->next;
    retval = (*head)->val;
    free(*head);
    *head = next_node;

    return retval;
}
```

Removing the last item

- Removing the last item from a list is very similar to adding it to the end of the list, but with one big exception - since we have to change one item before the last item, we actually have to look two items ahead and see if the next item is the last one in the list:

Removing the last item

אוניברסיטת אריאל בשומרון

- Removing the last item from a list is very similar to adding it to the end of the list, but with one big exception - since we have to change one item before the last item, we actually have to look two items ahead and see if the next item is the last one in the list:

```
int remove_last(node_t * head) {
    int retval = 0;
    /* if there is only one item in the list, remove it */
    if (head->next == NULL) {
        retval = head->val;
        free(head);
        return retval;
    }

    /* get to the second to last node in the list */
    node_t * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    /* now current points to the second to last item of the list, so let's remove current->next */
    retval = current->next->val;
    free(current->next);
    current->next = NULL;
    return retval;

}
```

Removing a specific item

אוניברסיטת אריאל בשומרון

- To remove a specific item from the list, either by its index from the beginning of the list or by its value, we will need to go over all the items, continuously looking ahead to find out if we've reached the node before the item we wish to remove. This is because we need to change the location to where the previous node points to as well.
- Here is the algorithm:
  - Iterate to the node before the node we wish to delete
  - Save the node we wish to delete in a temporary pointer
  - Set the previous node's next pointer to point to the node after the node we wish to delete
  - Delete the node using the temporary pointer

```c
int remove_by_index(node_t ** head, int n) {
    int i = 0;
    int retval = -1;
    node_t * current = *head;
    node_t * temp_node = NULL;

    if (n == 0) {
        return pop(head);
    }

    for (i = 0; i < n-1; i++) {
        if (current->next == NULL) {
            return -1;
        }
        current = current->next;
    }

    temp_node = current->next;
    retval = temp_node->val;
    current->next = temp_node->next;
    free(temp_node);

    return retval;

}
```

Variable Length Arrays (VLA)

- How to allocate an array with size defined “on the fly” ?

https://www.youtube.com/watch?v=JW3Vg0xpJLY&ab_channel=NesoAcademy

- VLAs disadvantages
  - VLAs allocate arrays on the stack, in runtime, making it harder, or even impossible to determine the stack size used at compile time.
  - Since the stack has a rather small amount of memory available (in comparison with the heap), many worry that VLAs have a great potential for stack overflow.

VLA Vs. Heaps

אוניברסיטת אריאל בשומרון

- Why to use VLA -
  - Fast - VLAs should be used when there are small amounts of local data, because they have fast allocation time and automatic clean-up.
  - Easy - a simple definition, no pointer to initialize, to check to free and no risk of memory leaks.
  - Readable - it's really a simple concept, so less likely to introduce subtle bugs.
- Why to use heaps -
  - Size limited - as already said, the stack can blow up. Dynamic arrays (heaps) should be used when there are large amounts of data, to prevent stack overflow.
  - Buffer overflows in VLA are a bit more serious than on heap memory (one can argue that it's an advantage, as a crashing application is better than a one silently corrupting data and eventually crashing on unrelated instructions).
  - Global - heaps should be used when the data needs to persist after the execution of a function and be available elsewhere in the program.
  - Portability - Dynamic arrays should be used when you have exceptional and/or irrational portability requirements.

אוניברסיטת
אריאל
בשומרון

Common memory bugs

1st bug – Memory leaks

אוניברסיטת אריאל בשומרון

- If a programmer fails to deallocate an unused region of memory, then the region will never be re-used. This wasted memory may eventually cause the application to exhaust memory resources on a system, eventually leading to failure or worse.

```c
/* Function with memory leak */
#include <stdlib.h>

void f()
{
    int* ptr = (int*)malloc(sizeof(int));

    /* Do some work */

    return; /* Return without freeing ptr*/
}
```

We should use free() after malloc() if memory is not used anymore.

```c
/* Function without memory leak */
#include <stdlib.h>

void f()
{
    int* ptr = (int*)malloc(sizeof(int));

    /* Do some work */

    free(ptr); // Deallocate memory
    return;
}
```

Solution for 1st bug

- In this approach, we will create two global counters and initialize them with 0.
- In every successful allocation, we will increment the value of the counter1 and after the deallocating the memory we will increment the counter2.
- In the end of the application, the value of both counters should be equal.

```
static unsigned int Allocate_Counter  = 0;

static unsigned int Deallocate_Counter  = 0;


void *Memory_Allocate (size_t size)
{
    void *pvHandle = NULL;

    pvHandle = malloc(size);
    if (NULL != pvHandle)
    {
        ++Allocate_Counter;
    }
    else
    {
        //Log error
    }
    return (pvHandle);
}
```

```
void Memory_Deallocate (void *pvHandle)
{
    if(pvHandle != NULL)
    {
        free(pvHandle);
        ++Deallocate_Counter;
    }
}
```

```
int Check_Memory_Leak(void)
{
    int iRet = 0;
    if (Allocate_Counter != Deallocate_Counter)
    {
        //Log error
        iRet = Memory_Leak_Exception;
    }
    else
    {
        iRet = OK;
    }
    return iRet;
}
```

2nd bug – no checking of malloc return

אוניברסיטת אריאל בשומרון

- It is a very common mistake.
- When we call the *malloc*, then it returns the pointer to the allocated memory.
- If there is no free space is available, the *malloc* function returns the NULL. It is good habits to verify the allocated memory because it can be NULL.
- If we try to dereference the **null pointer**, we will get an error.

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

int main(void)
{
    int *piBuffer = NULL;
    int n = 10, i = 0;

    //creating integer of size n.
    piBuffer = malloc(n * sizeof(int));

    //make sure piBuffer is valid or not
    if (piBuffer == NULL)
    {
        // allocation failed, exit from the program
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }

    //Assigned value to allocated memory
    for (i = 0; i < n; ++i)
    {
        piBuffer[i] = i * 3;
    }

    //Print the value
    for (i = 0; i < n; ++i)
    {
        printf("%d\n", piBuffer[i]);
    }

    //free up allocated memory
    free(piBuffer);

    return 0;
}
```

3rd bug – uninitialized memory

אוניברסיטת אריאל בשומרון

- Generally, c programmer uses malloc to allocate the block of memory.
- Some programmers assume that malloc allocated memory is initialized by the zero and they use the block of memory without any initialization.
- However, this assumption is wrong
- Solution – using *calloc()*

```
int * Foo(int *x, int n)
{
    int *piBuffer = NULL;
    int i = 0;

    //creating an integer array of size n.
    piBuffer = malloc(n * sizeof(int));

    //make sure piBuffer is valid or not
    if (piBuffer == NULL)
    {
        // allocation failed, exit from the program
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }

    //Add the value of the arrays
    for (i = 0; i < n; ++i)
    {
        piBuffer[i] = piBuffer[i] + x[i];
    }

    //Return allocated memory
    return piBuffer;
}
```

4th bug – access a freed memory

- When you freed the allocated memory then still pointer pointing to the same address.
- Dangling pointer - if you attempt to read or write the freed pointer - it is illegal and can be the cause of the code crashing (even if not has to).

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

int main()
{
    int *piData = NULL;

    piData = malloc(sizeof(int) * 10); //creating integer of size 10.

    free(piData); //free the allocated memory

    *piData = 10; //piData is dangling pointer

    return 0;
}
```

5th bug – re-free the same memory

אוניברסיטת אריאל בשומרון

- A *free()* function is used to deallocate the allocated memory.
- If an argument is pointing to a memory that has been already deallocated, the behavior of free function would be undefined.
- The freeing of the memory twice is more dangerous then memory leak, so it is very good habits to assigned the NULL to the deallocated pointer because the free function does not perform anything with the **null pointer**.

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

int main()
{
    int *piData = NULL;

    //creating integer of size 10.
    piData = malloc(sizeof(int) * 10);
    if(piData == NULL)
    {
        return -1;
    }

    //free the allocated memory
    free(piData);

    //free the allocated memory twice
    free(piData);

    return 0;
}
```

6th bug – free memory that was not allocated by memory functions

אוניברסיטת אריאל בשומרון

- The free function only deallocates the allocated memory. If a variable is not pointing to a memory that is allocated by the memory management function (e.g malloc, calloc) , the behavior of the free function will be undefined.

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

int main()
{
    int Data = 0;

    int *piData  = &Data;

    //free the memory
    free(piData);

    return 0;
}
```

7th bug – using malloc when is not needed

- Declaration of the normal array is easy and fast. The allocated memory of the normal array is automatically released by the compiler when the control comes out from the function.
- On the other hand, dynamic memory allocation is slow and only released by the developer explicitly to call the free function.
- So it is beneficial to use the normal array when the array is not needed after the function returns.

```c
void Foo(int n)
{
    int *piBuffer = NULL;

    //creating an integer array of size n.
    piBuffer = malloc(n * sizeof(int));

    //make sure piBuffer is valid or not
    if (piBuffer == NULL)
    {
        // allocation failed, exit from the program
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }

    free(piBuffer);
}
```

```c
void Foo(int n)
{
    int piBuffer[n]; //normal array
}
```

8th bug – sizeof for dynamic array size

- Some developers use the sizeof operator to calculate the size of the dynamically allocated array. The sizeof operator is used to calculate the size of static array - not used for the dynamic array. If you tried to calculate the size of the dynamic array then you will get the size of the pointer.

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

int main (void)
{
    int *piBuffer = NULL;
    int n = 10;

    //creating an integer array of size n.
    piBuffer = malloc(n * sizeof(int));

    //make sure piBuffer is valid or not
    if (piBuffer == NULL)
    {
        // allocation failed, exit from the program
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }

    printf("%d\n", sizeof(piBuffer));

    free(piBuffer);

    return 0;
}
```

Solution for 8th bug

- It is a great idea to carry the length of the dynamic array. Whenever you have required the length of the array, you need to read the stored length.
- To implement this idea in the program we need to allocate some extra space to store the length. It is my advice, whenever you use the technique then check that length of the array should not exceed the type of the array.
- Suppose you need to create an integer array whose size is n. So to carry the array length of the array, you need to allocate the memory for n+1

```
int *piArray = malloc ( sizeof(int) * (n+1) );
```

- If memory is allocated successfully, assign n (size of the array) its 0 places.

```
piArray[0] = n;
      or
* piArray = n;
```

- Now it's time to create a copy of the original pointer but to left one location from the beginning.

```
int * pTmpArray = piArray +1;
```

- Now, whenever in a program you ever required the size of the array then you can get from copy pointer.

```
ArraySize = pTmpArray[-1];
```