NANYANG TECHNOLOGICAL UNIVERSITY

CE1007/CZ1007 DATA STRUCTURES

Lecture 01: **Dynamic Data Structures**

Dr. Owen Noel Newton Fernando

College of Engineering

School of Computer Science and Engineering

OUTLINE

- What is dynamic data structure?
- Computer memory layouts
- malloc(): memory allocation in C
- free(): memory deallocation
- Common mistakes

YOU SHOULD BE ABLE TO...

- Explain the difference between static and dynamic elements
- Decide when to use a static or dynamic element
- Dynamically allocate an element in C
- Keep track of a dynamically allocated element (using a pointer)

OUTLINE

- **What is dynamic data structure?**
- Computer memory layouts
- malloc(): memory allocation in C
- free(): memory deallocation
- Common mistakes

WHAT IS A DYNAMIC DATA STRUCTURE?

- We will answer that question by looking at a "static" data structure
  - Data storage elements in C whose structure can't be changed once you write your program
- You have already encountered these
  - Arrays, C structs

WHAT IS A STATIC DATA STRUCTURE?

- You have already encountered:
  - int i;
  - char c;
  - char name[20];
  - struct account john={"OCBC Bank",1000.43, 4000.87};

WHAT IS A STATIC DATA STRUCTURE?

- You have already encountered:
  - int **i**;
  - char **c**;
  - char **name[20]**;
  - struct account **john**={"OCBC Bank",1000.43, 4000.87};
- Structure (including size) of these variables **cannot be changed** once you run the program.

WHAT IS THE TROUBLE FOR STATIC DATA STRUCTURE?

- Consider the following problem:

> Write a program that asks the user how many integers will be entered, then asks for each integer.

WHAT IS THE TROUBLE FOR STATIC DATA STRUCTURE?

```
1.  void main ()
2.  {       int n;
3.          int numArray[100];
4.          scanf("%d", &n);
5.          for (int i=0; i<n; i++){
6.                      scanf("%d", &numArray[i]);}
7.  }
```

What if the user inputs 101 for n?

WHAT IS THE TROUBLE FOR STATIC DATA STRUCTURE?

```
1.  void main ()
2.  {       int n;
3.          int numArray[101];
4.          scanf("%d", &n);
5.          for (int i=0; i<n; i++){
6.                      scanf("%d", &numArray[i]);}
7.  }
```

What if the user inputs 200 for n?

WHAT IS THE TROUBLE FOR STATIC DATA STRUCTURE?

```
1.  #define MAX_NUM 10000
2.  void main ()
3.  {       int n;
4.          int numArray[MAX_NUM];
5.          scanf("%d", &n);
6.          for (int i=0; i<n; i++){
7.                      scanf("%d", &numArray[i]);}
8.  }
```

No matter how you define MAX_NUM, it might still be insufficient.
What's more, it will waste space.

CAN I DO THIS?

```
1.  void main ()
2.  {       int n;
3.          scanf("%d", &n);
4.          int numArray[n];
5.          for (int i=0; i<n; i++){
6.                  scanf("%d", &numArray[i]);}
7.  }
```

No, the C complier will not cooperate. It needs to know the exact size of space to set aside for the array.

CAN I DO THIS

- Problems
  - Have to declare array size before compilation
  - Compiler needs to know how much space to set aside for the array

```
scanf("%d", &numOfNumbers);
int numArrays[numOfNumbers]; //Not allowed
```

  - Cannot change array size while code is running

- Solution so far
  - Just pick some big number for array size
    - int numbers[10000]
  - Ignore the wasted space

STATIC VARIABLES

- All declared at compile-time
  - If you want 3 structs, declare 3 separate structs with 3 unique names in code

```
struct mystruct s1, s2, s3;
```

  - Note that even with the array declaration,

```
struct mystruct s_arr[3]
```

    each struct has a distinct "name" that you use to access it

```
s_arr[0], s_arr[1], s_arr[2]
```

- What if you want to declare more variables/arrays/structs when your code is already running?

DYNAMIC VARIABLES

```
1.  void main ()
2.  {       int n;
3.          scanf("%d", &n);
4.          // dynamically declare int array of size n
5.          for (int i=0; i<n; i++){
6.                      scanf("%d", &numArray[i]);}
7.  }
```

Can we do that with dynamic variables?

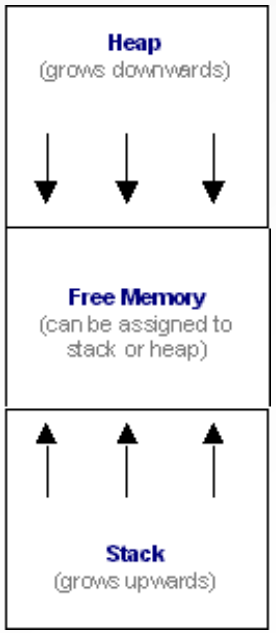
Yes!!

OUTLINE

- What is a dynamic data structure?
- **Computer memory layouts**
- malloc(): memory allocation in C
- free(): memory deallocation
- Common mistakes

HOW?

- First, we need to know the layout of computer memory



HOW ARE ELEMENTS LAID OUT IN MEMORY?

- Static vs dynamic memory
- Elements in "static" memory are allocated on the stack

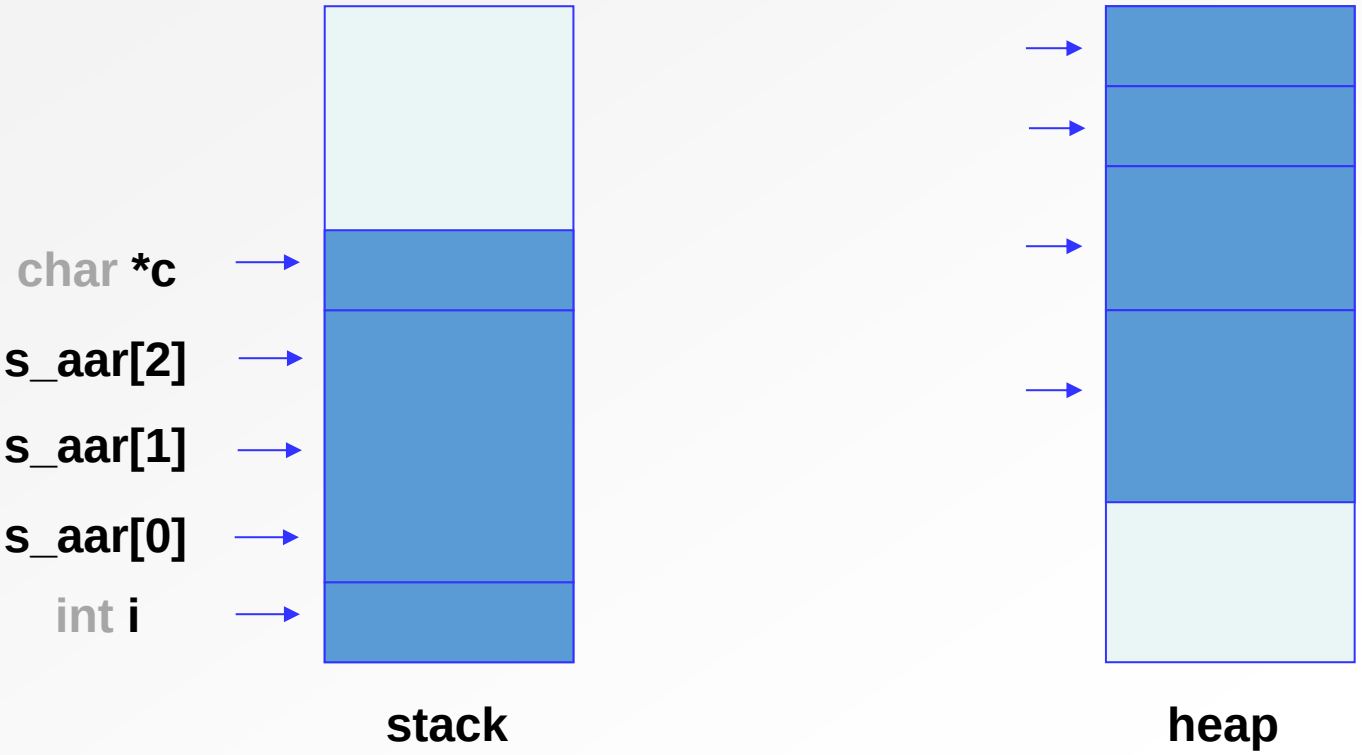
```
void main(){
        int i;
        char c;
        struct mystruct s_arr[3];
}
```

- Elements in "dynamic" memory are allocated on the heap
  - This is what we'll learn to do in a few more slides
  - You'll be doing a lot of this with data structures

HOW ARE ELEMENTS LAID OUT IN MEMORY?

- **Static** variables – **stack**
  Elements are nicely stacked on top of one another

- **Dynamic** variables – **heap**
  Elements can be allocated anywhere **during run-time**



OUTLINE

- What is a dynamic data structure?
- Computer memory layouts
- ***malloc()*: memory allocation in C**
- free(): memory deallocation
- Common mistakes

DYNAMIC MEMORY ALLOCATION

"Dynamic" memory allocation refers to allocation of memory during program execution time, rather than during compile time.

- Utility functions such as malloc() are provided in the standard library to allocate memory dynamically.
  - malloc() stands for "memory allocation".
  - Memory allocated by malloc() is not cleared

malloc()

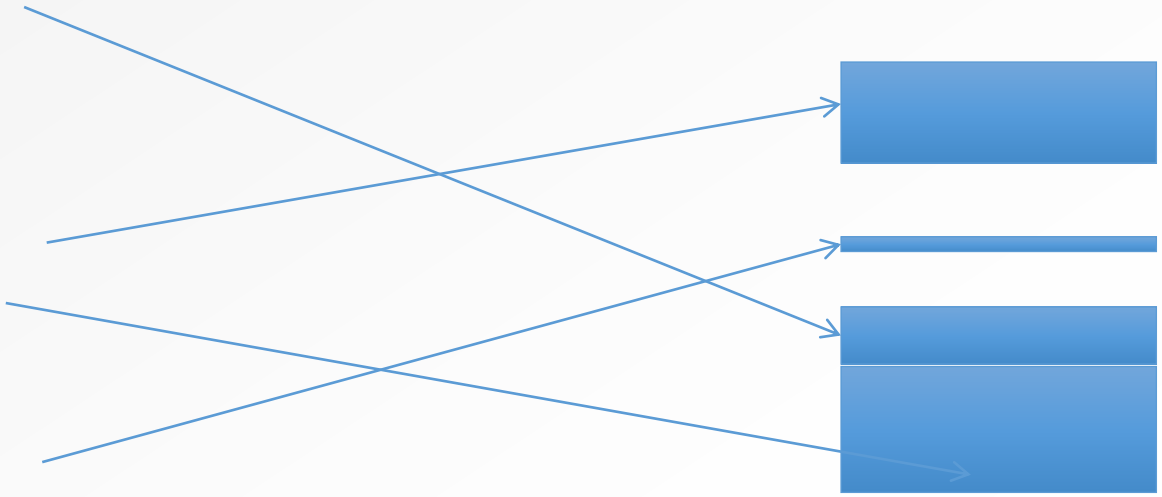
- C provides a function for memory allocation on the heap during run-time

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

- Reserves *size* bytes of memory for your variable/ structure, **e.g., int *i=malloc(sizeof(int)) ;**
- We use the **sizeof()** to pass in the correct number of bytes. (ensure correct size on different platforms) ;
- Returns the address (a **pointer**) where the reserved space starts
  - Returns NULL if memory allocation fails
- Fun question: what is a **(void *)**?
  - Think about it… search for answers from help in IDE, textbooks and google

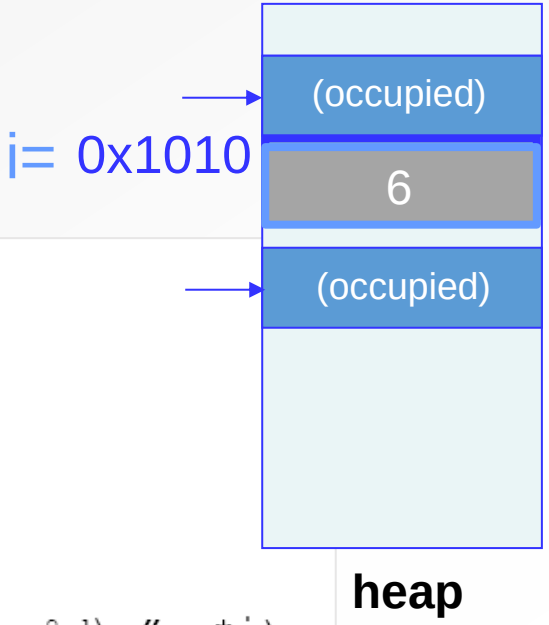
malloc()

- Each time you call malloc(size), the OS (not the compiler) looks for a space in the heap with size contiguous bytes of memory
  - One way that malloc() can fail is if your memory is very fragmented
  - Many small blocks free, but none are large enough to fit size bytes



malloc() BASICS: INT

- Notice that we no longer have to declare an integer i, but we still need a pointer to keep track of the allocated memory




```
1  #include <stdlib.h>
2  int main(){
3      int *i;
4      i = malloc(sizeof(int));
5      if (i == NULL)
6          printf("Uh oh.\n");
7      scanf("%d", i);
8      printf("The magic number is %d\n", *i);
9  }
```

What is the output?

BACK TO OUR PROBLEM

```
1.  void main ()
2.  {       int n;
3.          scanf("%d", &n);
4.          // dynamically declare int array of size n
5.          for (int i=0; i<n; i++){
6.                      scanf("%d", &numArray[i]);}
7.  }
```

Can we do that with dynamic variables?

Yes!!

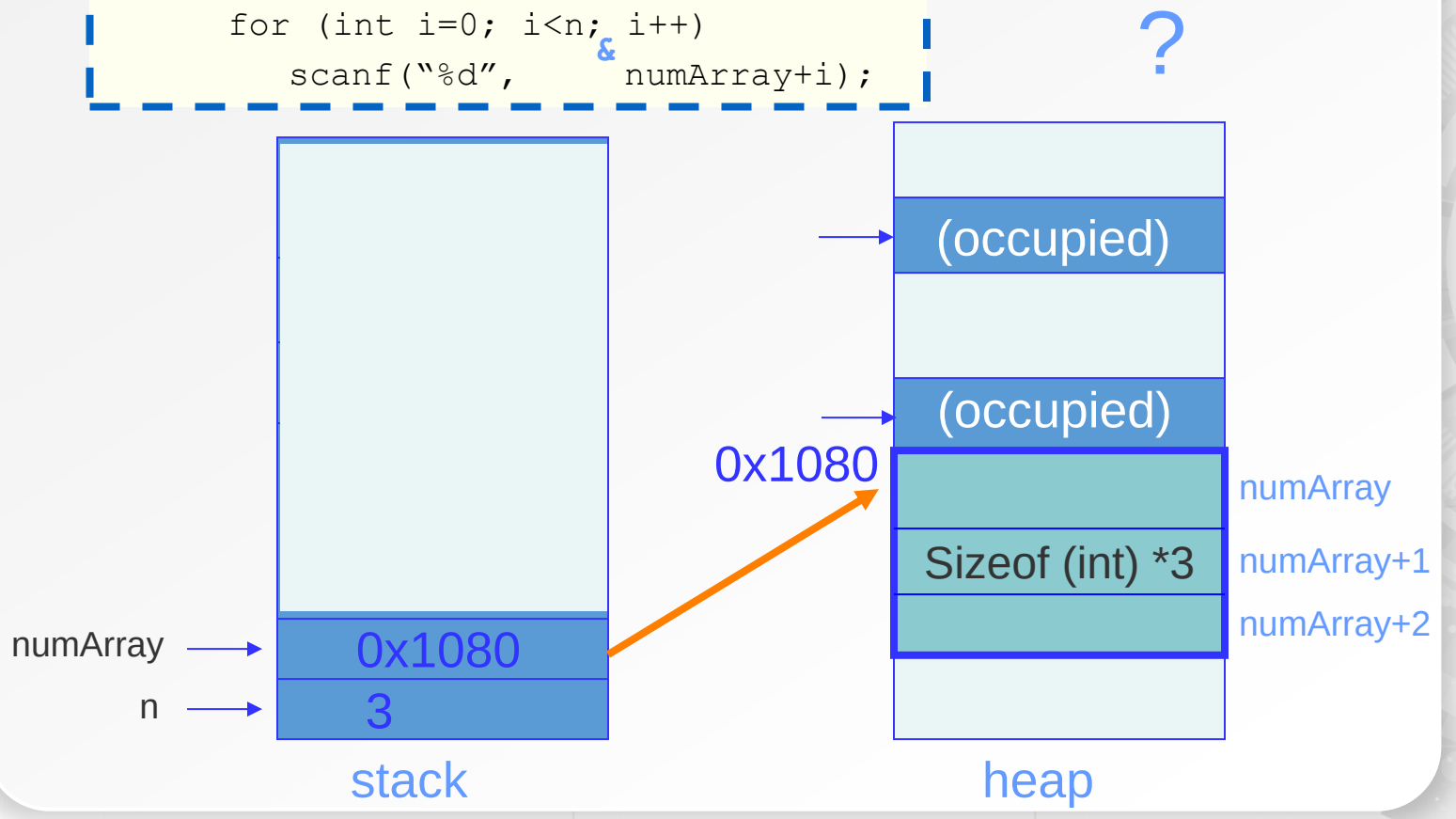
malloc() BASICS: INT ARRAY

- Again, pointer to keep track of array
  - Stores address of start of the array
  - i.e., pointer takes you to the first element
- Allocate exactly the right sized array after n is entered.
- Size to allocate = number of elements * sizeof(each element)

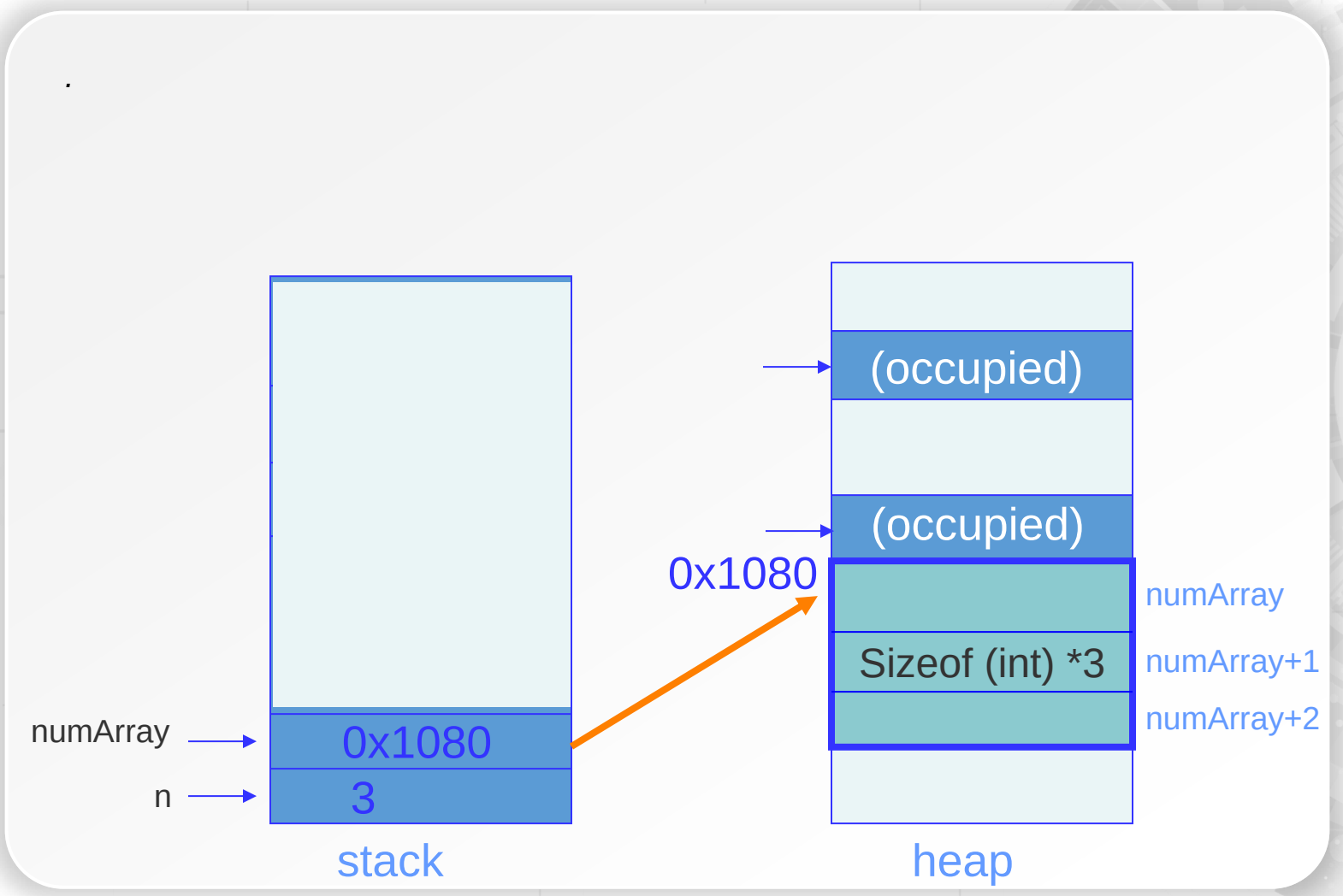
```
#include <stdlib.h>

void main ()

{  int n;
   int *numArray;
   scanf("%d", &n);
   numArray = malloc(sizeof(int)*n);
   if (numArray!=NULL)
      for (int i=0; i<n; i++){
         scanf("%d", numArray+i);}
}
```

malloc() BASICS: INT ARRAY

```
int n;
int *numArray;
scanf("%d", &n);
numArray = malloc(sizeof(int)*n);
 if (numArray!=NULL)
    for (int i=0; i<n; i++)
       scanf("%d", &numArray+i);
```



malloc() BASICS: INT ARRAY



MORE EXAMPLES OF malloc()

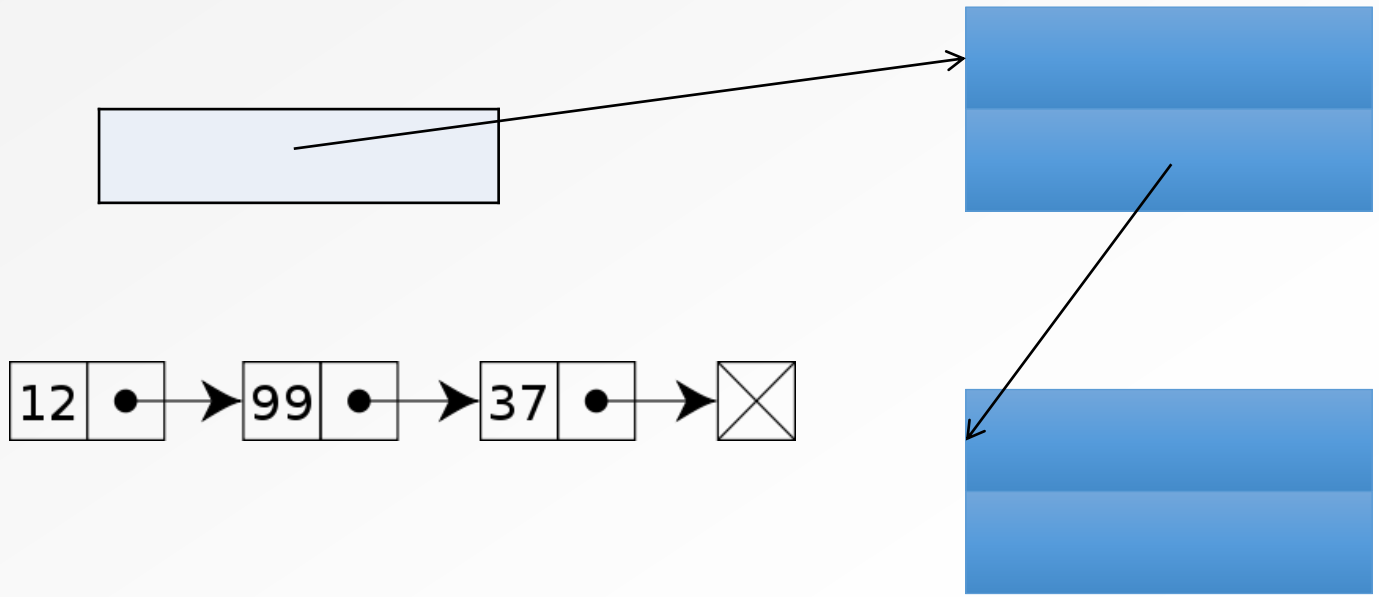
malloc() FOR STRING/CHAR ARRAY

- Same as int array, except that it is char (**sizeof(char)=1**)
- For a string with length n, allocate n+1 bytes, with 1 for string terminator '\0'
- Question: what if you allocate 10 chars but enter an 11-char string? (will talk about it in 'common mistakes')

```
1   #include <stdlib.h>
2   int main(){
3       int n;
4       char *str;
5       printf("How long is your string? ");
6       scanf("%d", &n);
7       str = malloc(n+1);
8       if (str == NULL) printf("Uh oh.\n");
9       scanf("%s", str);
10      printf("Your string is: %s\n", str);
11  }
```

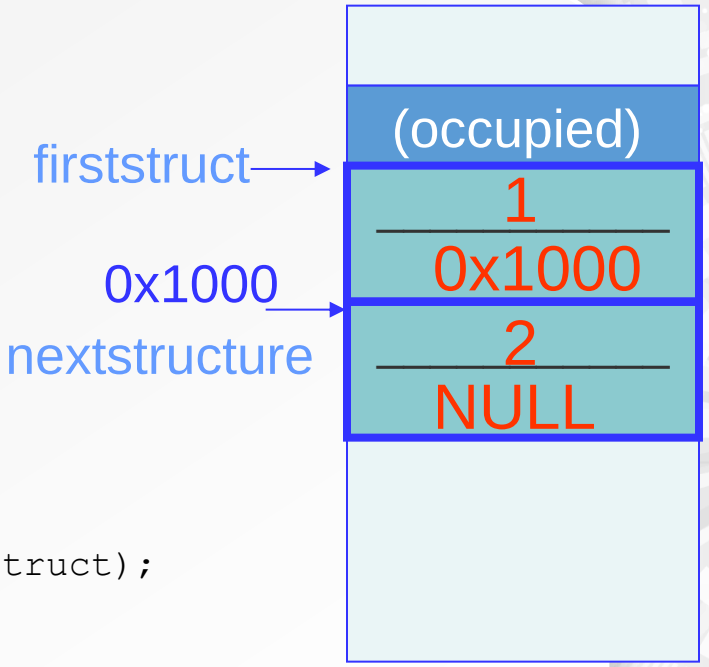
MALLOC() BASICS: STRUCT TO STRUCT

- Let’ s make this more complicated
- So far, we malloc() some memory and point to it using a named (static) variable
- What if we take a dynamically allocated pointer variable and point it to another dynamically allocated element?
- Let’ s figure out the concept before we look at code



malloc() FOR STRUCTURE

```
1.  #include <stdlib.h>
2.  struct mystruct{
3.        int number;
4.        struct mystruct *nextstruct;
5.  };
6.  int main() {
7.  struct mystruct *firststruct;
8.  firststruct = malloc(sizeof(struct mystruct);
9.  firststruct->number = 1;
10. firststruct->nextstruct = malloc(sizeof(struct mystruct));
11. firststruct->nextstruct->number = 2;
12. firststruct->nextstruct->nextstruct = NULL;
13. }
```



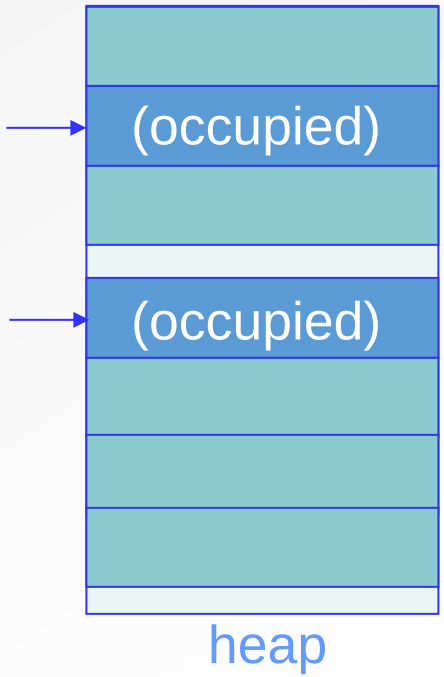
We will focus on this in the linked list lectures!

OUTLINE

- What is a dynamic data structure?
- Computer memory layouts
- malloc(): memory allocation in C
- ***free()*: memory deallocation**
- Common mistakes

WHAT HAPPENS IF MYFUNC() IS CALLED 10000000 TIMES?

```
#include <stdlib.h>
void myfunc(){
   int *i = malloc(sizeof(int));
   …
}
void main(){
   for (int i=0;i<10000000;i++)
            myfunc();
}
```



- Elements created using malloc() are not automatically cleared
- When memory is continually being dynamically allocated but not cleared, the computer eventually runs out of memory
- We need a way to free up dynamically allocated elements once we are done with them

free()

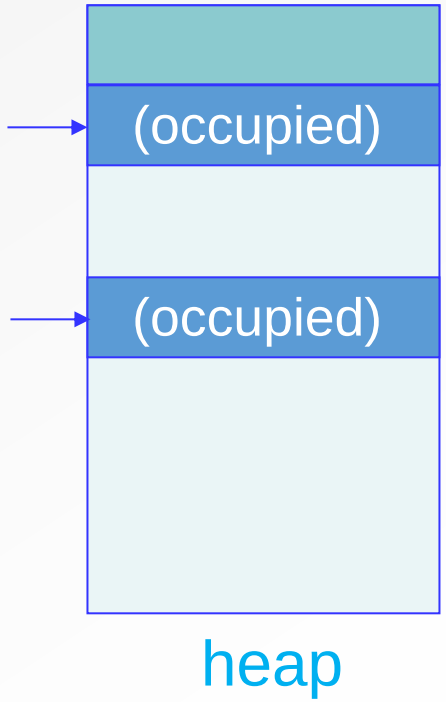
- The free() function allows you to clear up memory when you are done with the element

```
free(pointer);
```

  - **free()** does **not** need to know how many bytes to clear
  - System keeps track of size of each block you allocated using **malloc()**

WHAT HAPPENS IF MYFUNC() IS CALLED 10000000 TIMES?

```
#include <stdlib.h>
void myfunc(){
  int *i = malloc(sizeof(int));
     …
  free (i);
}
void main(){
  for (int i=0;i<10000000;i++)
           myfunc();
}
```



No problem for the memory now!

OUTLINE

- What is a dynamic data structure?
- Computer memory layouts
- malloc(): memory allocation in C
- free(): memory deallocation
- **Common mistakes**

EXPLICIT TYPE CASTS

- If you **forget to include stdlib.h**, you will probably get a warning message

```
source1.c(10): warning C4013: 'malloc'
undefined; assuming extern returning int
```

- If you name the code file as **.cpp**, you will get an error message. Make sure your code file name is **.c**

```
source1.cpp(10):
error C3861: 'malloc':
identifier not found
```

```
#include <stdlib.h>
void myfunc()
{
   int *i = malloc(sizeof(int));
      …
   free (i);
}
```

EXPLICIT TYPE CASTS

```
#include <stdlib.h>
void myfunc()
{
    int *i = malloc(sizeof(int));
    …
    free (i);
}
```

This is the standard way to do!

- For c, if you include stdlib.h, the compiler should automatically take care of everything

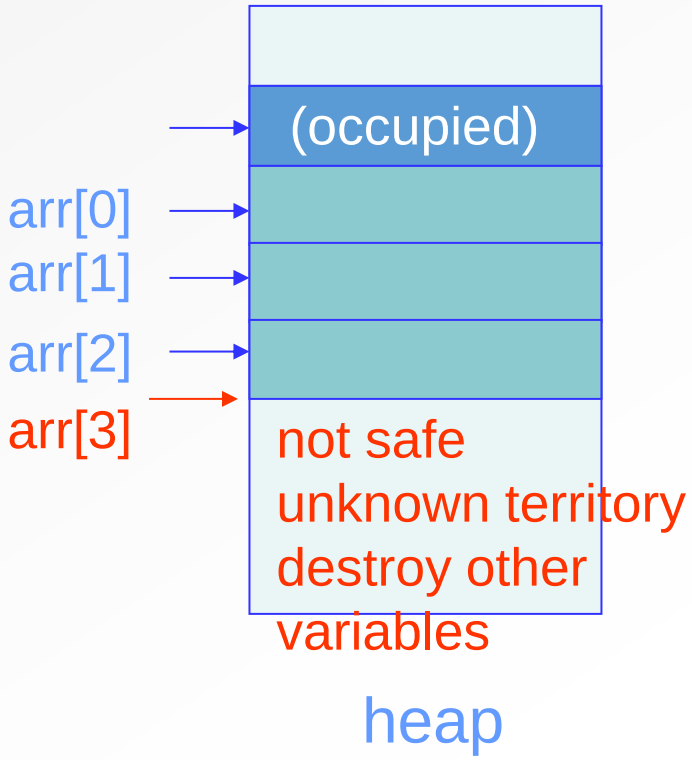
```
void myfunc()
{
    int *i=(int *)malloc(sizeof(int));
    …
    free (i);
}
```

- If you explicitly introduce a type cast, the compiler will assume you know what you're doing, even if you make a mistake with the pointer type
- For cpp, even if you have included stdlib.h, you still need to put a type cast in front.

BUFFER OVERFLOWS

- Question: I used malloc(3 * sizeof(int)) to allocate space for an array of 5 integers and it works. Why?

```
#include <stdlib.h>
int main(){
    int i;
    int *arr = malloc(3 *
        sizeof(int));
    for (i=0; i<5; i++)
        arr[i] = i;
    for (i=0; i<5; i++)
        printf("%d ", arr[i]);
}
```

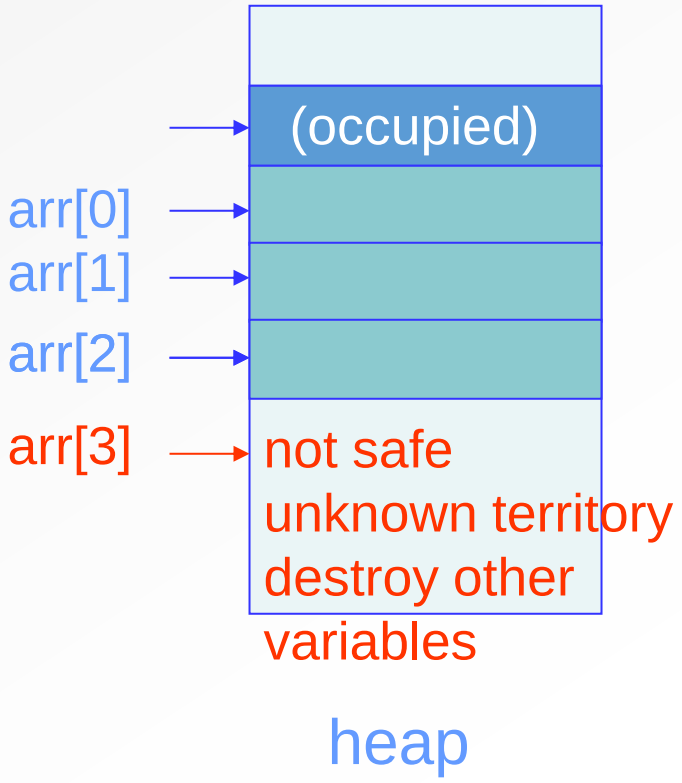


BUFFER OVERFLOWS

- Question: I used malloc(3 * sizeof(int)) to allocate space for an array of 5 integers and it works. Why?

Answer:

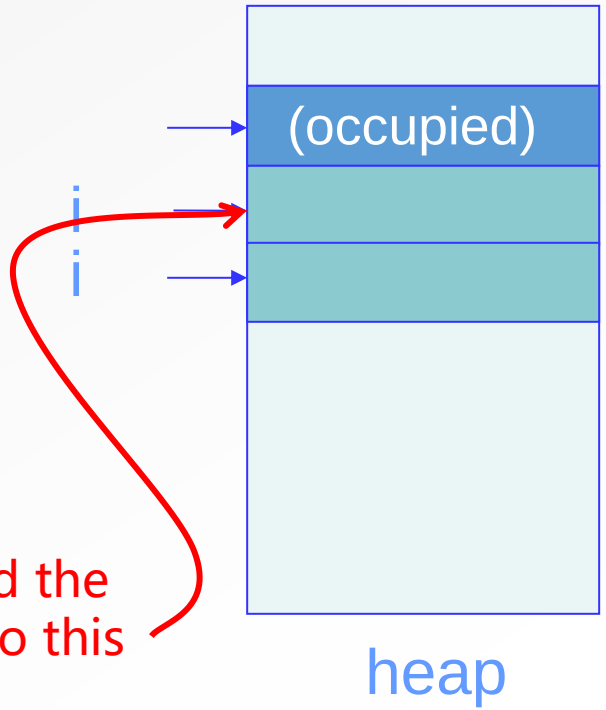
- **You have overwritten parts of memory that you were not supposed to**
- **These parts might store other variables or other program instructions**
- **Most of the time, this will crash your program**
- **But it might work if you are lucky**



MEMORY LEAKS

- When you allocate memory and then make it inaccessible, you have a memory leak
- This is very Bad.

```
1. #include <stdlib.h>
2. void main(){
3.    int *i;
4.    i = malloc(sizeof(int));
5.    i = malloc(sizeof(int));
6. }
```



After i=malloc(sizeof(int)) is called the second time, no one is pointing to this block of memory

TODAY: YOU SHOULD BE ABLE TO...

- Explain the difference between static and dynamic elements
- Decide when to use a static or dynamic element
- Dynamically allocate an element in C
- Keep track of a dynamically allocated element (using a pointer)

NEXT LECTURE

- We will have more fun on Linked List
- Use **malloc()** to create nodes for the linked list

ON BEING READY TO LEARN

- Growth vs fixed mindset
  - Individuals with a *fixed mindset* believe that their intelligence is an inborn trait—they have a certain amount, and that's that.
  - Individuals with a *growth mindset* believe that they can develop their intelligence over time.

Blackwell, Trzesniewski, & Dweck, 2007
Dweck, 1999, 2007

ON BEING READY TO LEARN

- These two mindsets lead to different school behaviors

"For one thing, when students view intelligence as fixed, they tend to value looking smart above all else. They may sacrifice important opportunities to learn—even those that are important to their future academic success—if those opportunities require them to risk performing poorly or admitting deficiencies.

Students with a growth mindset, on the other hand, view challenging work as an opportunity to learn and grow."

ON BEING READY TO LEARN

- Recommended reading
  - http://www.ascd.org/publications/educational-leadership/sept10/vol68/num01/Even-Geniuses-Work-Hard.aspx