NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CE1107/CZ1107: DATA STRUCTURES AND ALGORITHMS

Dynamic Data Structures

College of Engineering
School of Computer Science and Engineering

TODAY

- What is a data structure?
- Dynamic data structures
- Computer memory layouts
- Memory allocation in C
- Worked examples
- Memory deallocation
- Common mistakes

LEARNING OBJECTIVES

After this lesson, 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)

TODAY

- **What is a data structure?**
- Dynamic data structures
- Computer memory layouts
- Memory allocation in C
- Worked examples
- Memory deallocation
- Common mistakes

WHAT IS A DATA STRUCTURE?

- Very simply, a particular arrangement of data
- Why do we care?
  - Each arrangement allows you to do some things efficiently… and other things less efficiently
- Solving a given problem efficiently requires
  - The right algorithm(s) + right data structure(s)
- You'll learn more about this in algorithms

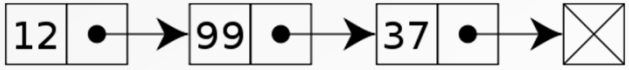
ABSTRACT DATA TYPES

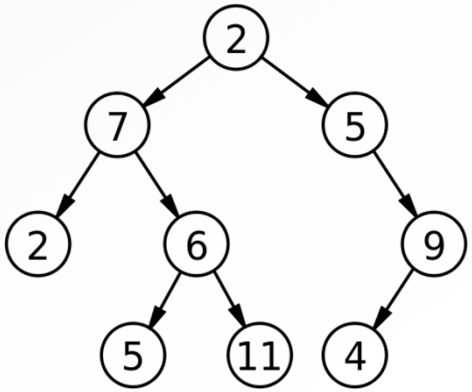
- You may encounter the term “abstract data type”
- Compare with “primitive”/”built-in” data types in C
  - int, float, double, char, etc.
- Abstract data types are defined according to what they can do
  - Conceptually, a stack data structure allows you to “push” new data to the top
  - Independent of the actual implementation (in some language X)

LINEAR VS. NON-LINEAR DATA STRUCTURES

- Linear data structures
  - Arranged sequentially, similar to an array
  - Good for storing sequential data such as lists

- Non-linear data structures
  - Sometimes, the data requires a hierarchical layout
  - Can also be used to store sequential data in a smart way (we'll see this in Week 5)





TODAY

- What is a data structure?
- **Dynamic data structures**
- Computer memory layouts
- Memory allocation in C
- Worked examples
- Memory deallocation
- Common mistakes

WHAT IS A DYNAMIC DATA STRUCTURE?

- We'll 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've already encountered these
  - Arrays, C structs

WRITE A PROGRAM

- Write a program that asks the user how many integers will be entered, then asks for each integer.
- What was challenging about this exercise?

BACK TO THE LABS

- Example code

```
1   #define MY_FAVORITE_BIG_NUMBER 100
2
3   int numOfNumbers;
4   int numArray[MY_FAVORITE_BIG_NUMBER];
5
6   scanf(“%d”, &numOfNumbers);
7   for (i=0; i<numOfNumbers; i++){
8           scanf(“%d”, &numArray[i]);
9
10  }
```

BACK TO THE LABS

- 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[1000]
  - Ignore the wasted space

STATIC VARIABLES

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

```
struct mystruct s1, s2, s3;
struct mystruct s_arr[3];
```

  - Note that even with the array declaration, 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

- What we want to be able to do

```
1   int numOfNumbers;
2
3   scanf("%d", &numOfNumbers);
4
5   //Declare int array of size numOfNumbers
6
7   for (i=0; i<numOfNumbers; i++){
8           scanf("%d", &numArray[i]);
9
10  }
```

DYNAMIC VARIABLES

- Not limited to array of integers
- What about declaring a new struct?

```
1   struct mystruct{int theinteger};
2
3   input = 0;
4   while (input != -1)
5            scanf("%d", &input);
6
7            // Declare a new struct and store the
8            // input value into theinteger
9            // Now store the new struct somewhere
10  }
```

TODAY

- What is a data structure?
- Dynamic data structures
- **Computer memory layouts**
- Memory allocation in C
- Worked examples
- Memory deallocation
- Common mistakes

HOW ARE ELEMENTS LAID OUT IN MEMORY?

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

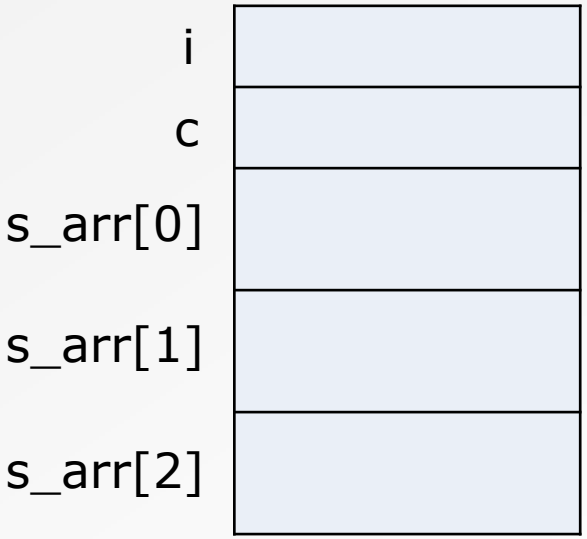
```
void main(){
    int i;
    char c;
    struct mystruct s_arr[10];
}
```

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

HOW ARE ELEMENTS LAID OUT IN MEMORY?

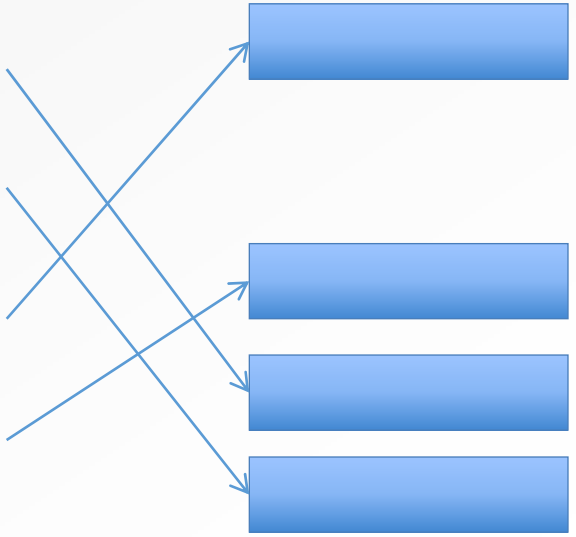
Stack

- Elements are nicely stacked on top of one another



Heap

- Memory space for elements can be allocated anywhere



TODAY

- What is a data structure?
- Dynamic data structures
- Computer memory layouts
- **Memory allocation in C**
- Worked examples
- Memory deallocation
- Common mistakes

DYNAMIC MEMORY ALLOCATION

- Earlier,

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

  - The compiler needs to know how many bytes of memory to allocate on the stack for each variable
  - On the heap, you can dynamically (at run-time) allocate any amount you want

- C provides a function or memory allocation on the heap

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

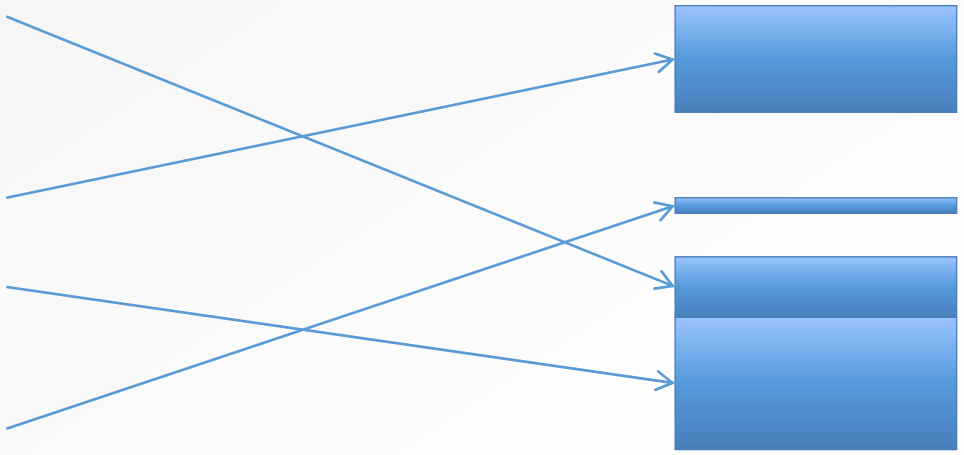
DYNAMIC MEMORY ALLOCATION

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

- Reserves *size* bytes of memory for your variable/structure
- 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… We'll talk about this at the start of the next lecture

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



TODAY

- What is a data structure?
- Dynamic data structures
- Computer memory layouts
- Memory allocation in C
- **Worked examples**
- Memory deallocation
- Common mistakes

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
- We use the sizeof() macro to pass in the correct number of bytes
  - Easier to ensure correct size passed in when compiling on different platforms

```
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   }
```

Note lines 3-4 can be written (a bit more confusingly) as:

```
int *i = malloc(sizeof(int));
```

malloc() BASICS: int ARRAY

- Again, pointer to keep track of array
  - Stores address of start of array
  - i.e., pointer takes you to first element
- Size to allocate = number of elements * sizeof (each element)
- Notice we allocate exactly the right sized array after we find out how many numbers will be entered

```
1   #include <stdlib.h>
2   int main(){
3       int n;
4       int *int_arr;
5       printf("How many integers do you have?");
6       scanf("%d", &n);
7       int_arr = malloc(n * sizeof(int));
8       if (int_arr == NULL) printf("Uh oh.\n");
9
10      // Loop over array and store integers entered
11  }
```

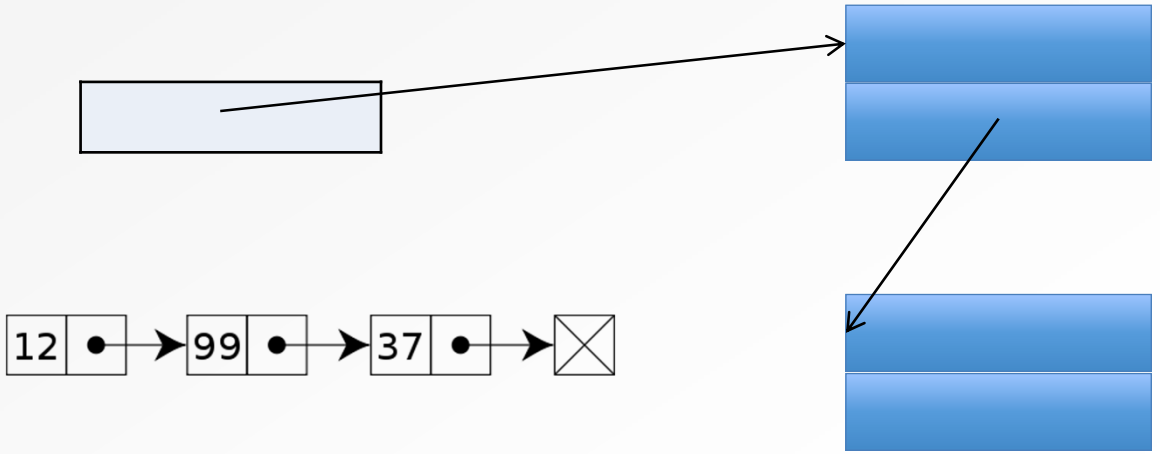
malloc() BASICS: string/char ARRAY

- Same as int array, except it's chars
- Allocate n+1 bytes to account for the \0 string terminator
- Fun question: What if you allocate 10 chars but enter an 11 char string?
  - We'll look at this in a few more slides

```
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

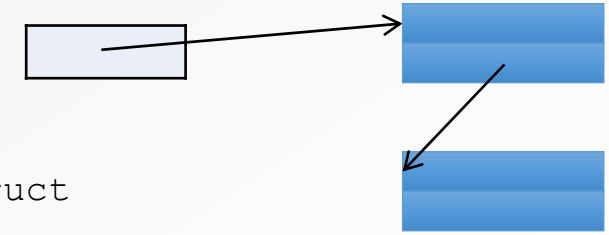
- Let's make this more complicated
- So far, we malloc() some memory and point to it using a named (static) variable
- What if we take a dynamically allocated pointer variable and point it to another dynamically allocated element?
- Let's figure out the concept before we look at the code



malloc() BASICS: struct TO struct

- **Hands-on:**
  - Before looking at the code on the next slide, try writing the code for yourself

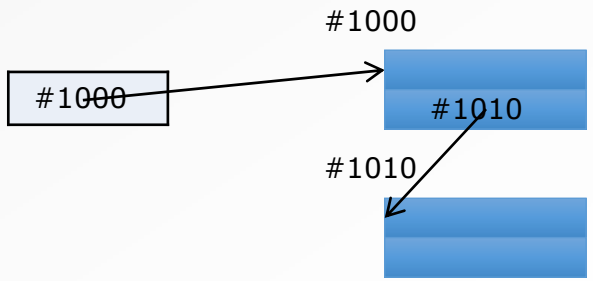
```
#include <stdlib.h>
 struct mystruct{
     int number;
     struct mystruct *nextstruct
 };
 int main(){




}
```



malloc() BASICS: struct TO struct

- Only the first struct is accessed through a statically declared element
- The second struct is linked to the first struct using the nextstruct pointer
- We'll be doing for the next four weeks – better get used to it!

```
1   #include <stdlib.h>
2   struct mystruct{
3       int number;
4       struct mystruct *nextstruct;
5   };
6
7   int main(){
8       struct mystruct *firststruct;
9
10      firststruct = malloc(sizeof(struct mystruct));
11      firststruct->number = 1;
12      firststruct->nextstruct = malloc(sizeof(struct mystruct));
13
14      firststruct->nextstruct->number = 2;
15      firststruct->nextstruct->nextstruct = NULL;
16  }
```



TODAY

- What is a data structure?
- Dynamic data structures
- Computer memory layouts
- Memory allocation in C
- Worked examples
- **Memory deallocation**
- Common mistakes

MEMORY DE-ALLOCATION

- Elements you create using malloc() are not automatically cleared
- We need a way to free up dynamically allocated elements once we're done with them
- What happens if myfunc() is called 10000000 times?

```
1   #include <stdlib.h>
2   int main(){
3       int i;
4       for (i=0; i<10000000; i++)
5           myfunc();
6   }
7
8   void myfunc(){
9       int *i = malloc(sizeof(int));
10  }
```

free()

- When memory is continually being dynamically allocated but not cleared, the computer eventually runs out of memory
- The free() function allows you to clear up memory when you're done with the element

```
free(ptr);
```

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

```
1   #include <stdlib.h>
2   int main(){
3       myfunc();
4   }
5
6   void myfunc(){
7       int *i = malloc(sizeof(int));
8       free(i);
9   }
```

TODAY

- What is a data structure?
- Dynamic data structures
- Computer memory layouts
- Memory allocation in C
- Worked examples
- Memory deallocation
- **Common mistakes**

EXPLICIT TYPE CASTS

- If you forget to include stdlib.h, you will probably get a warning like "assignment of pointer from integer lacks a cast"
- Example

```
Char *c = malloc(100);
```

- Some of you will fix this by writing

```
Char *c = (char *)malloc(100);
```

- 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 types
- If you use the first syntax and include stdlib.h, the compiler should automatically take care of everything

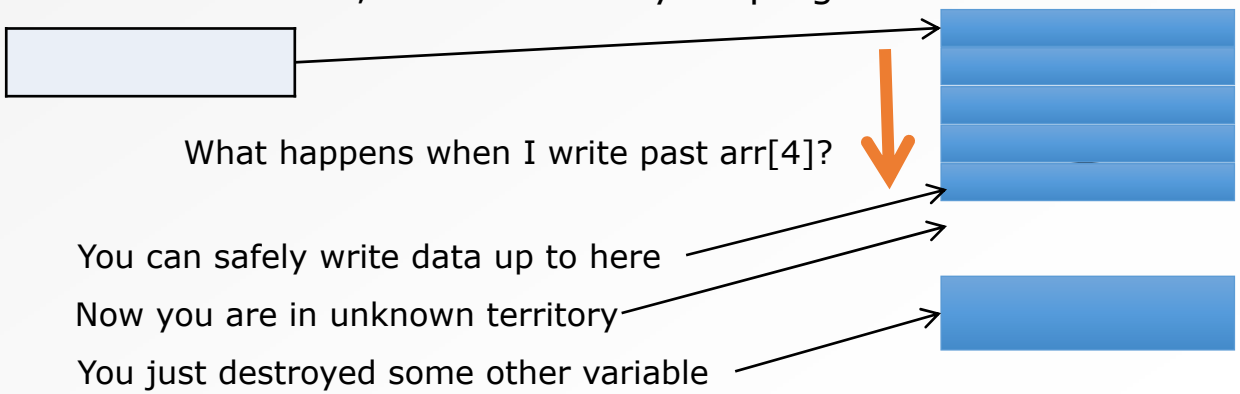
BUFFER OVERFLOWS

- Question: I used malloc(5 * sizeof(int)) to allocate space for an array of 10 integers and it works. Why?

```
1   #include <stdlib.h>
2   int main(){
3       int i;
4       int *arr = malloc(5 * sizeof(int));
5
6       for (i=0; i<10; i++)
7           arr[i] = i;
8
9       for (i=0; i<10; i++)
10          printf("%d ", arr[i]);
11  }
```

BUFFER OVERFLOWS

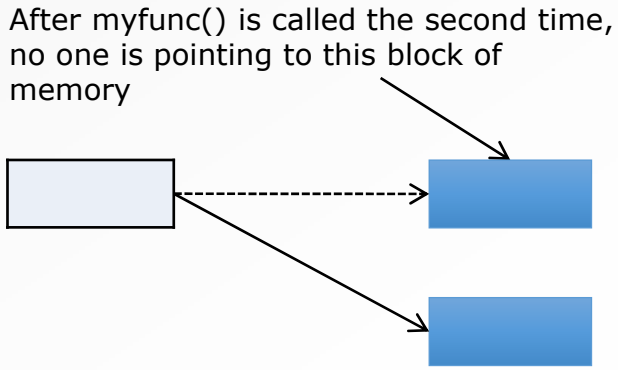
- Question: I used malloc(5 * sizeof(int)) to allocate space for an array of 10 integers and it works. Why?
- Answer:
  - You are lucky
  - You have overwritten parts of memory that you were not supposed to
  - These parts might store other variables or other program instructions
  - Most of the time, this will crash your program



MEMORY LEAKS

- When you allocate memory and then make it inaccessible, you have a memory leak
- This is very very very very very bad. Very. Bad.
- Everyone will do it at least once in their programming career

```
1   #include <stdlib.h>
2   int *i;
3   int main(){
4       myfunc();
5       myfunc();
6   }
7
8   void myfunc(){
9       *i = malloc(sizeof(int));
10  }
```

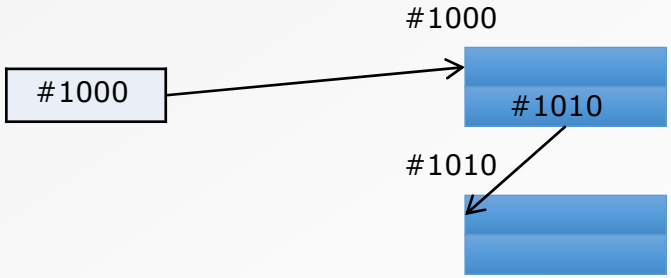


TODAY

- What is a data structure?
- Dynamic data structures
- Computer memory layouts
- Memory allocation in C
- Worked examples
- Memory deallocation
- Common mistakes

NEXT TIME

- Linked lists
  - Formalise what we did in struct->struct
  - One of the most fundamental data structures



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 behaviours

"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