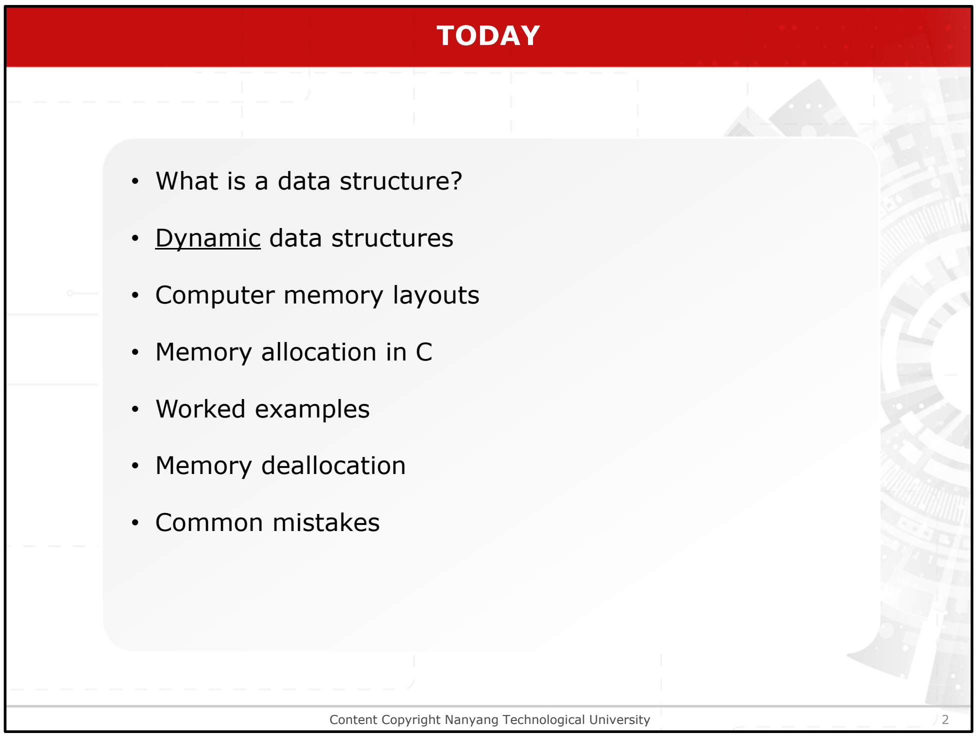
NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CE1107/CZ1107: DATA STRUCTURES AND ALGORITHMS

Dynamic Data Structures

College of Engineering
School of Computer Science and Engineering



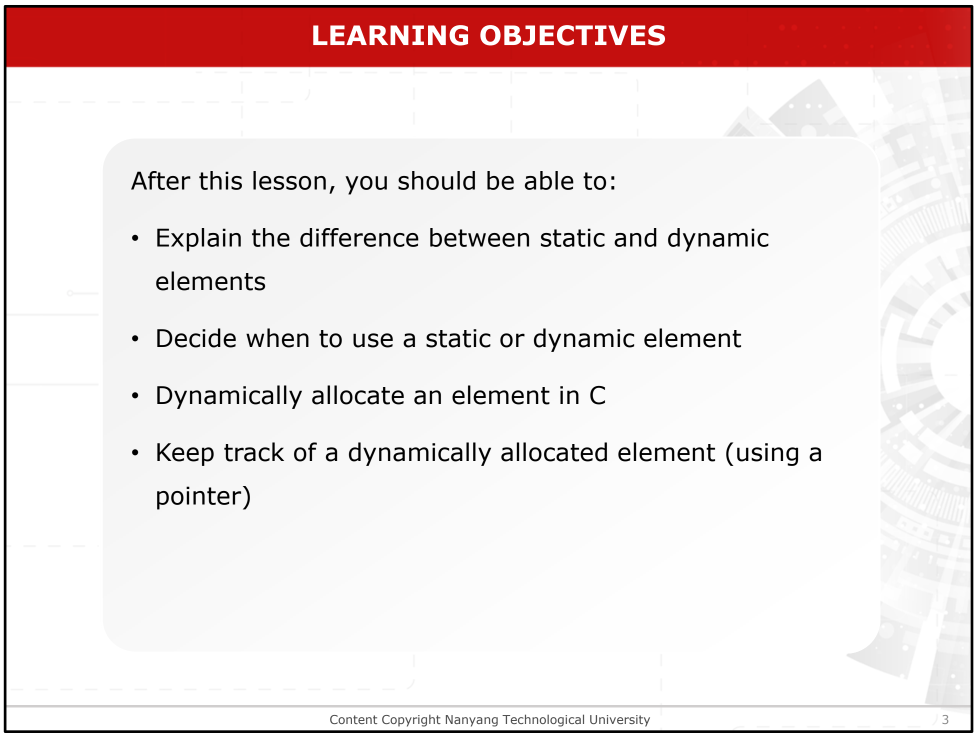
TODAY

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

2

Today, we will learn;

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



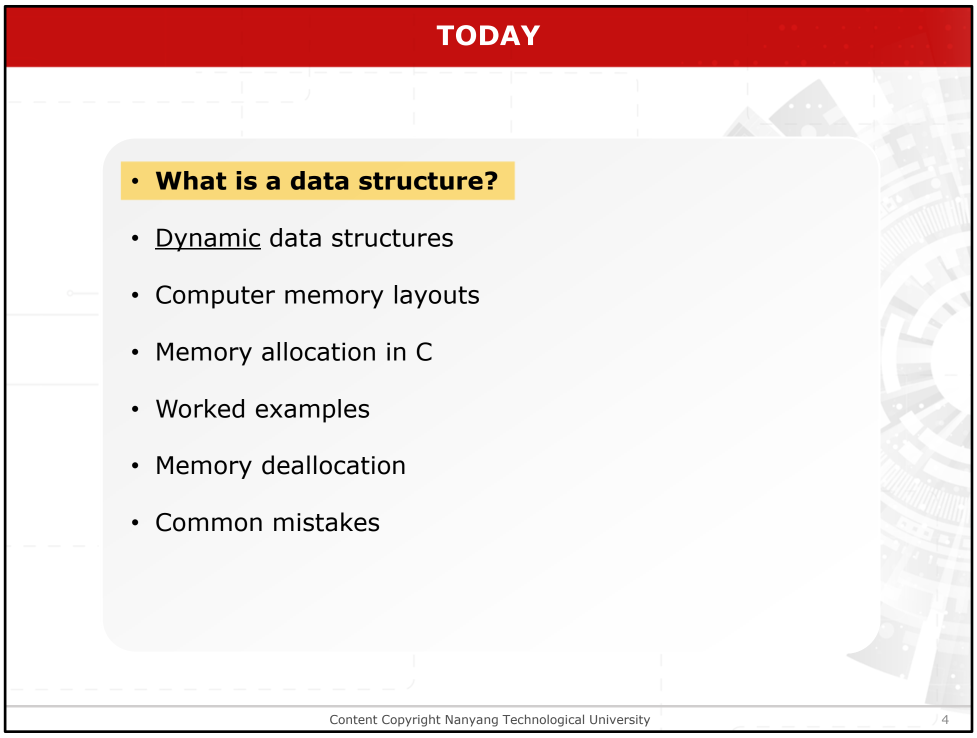
LEARNING OBJECTIVES

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)

3

- Understand the difference between a static and a dynamic data structure, or a static and a dynamic element.
- Use malloc() function to allocate memory in C dynamically.
- Understand the use of pointers in dynamic memory allocation

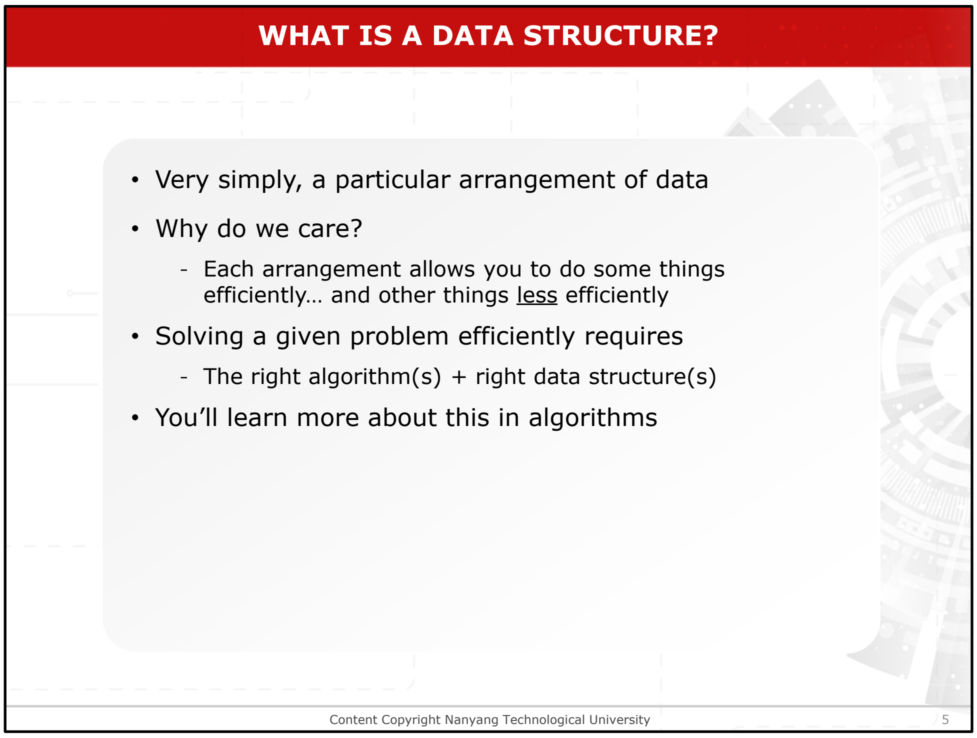


TODAY

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

4

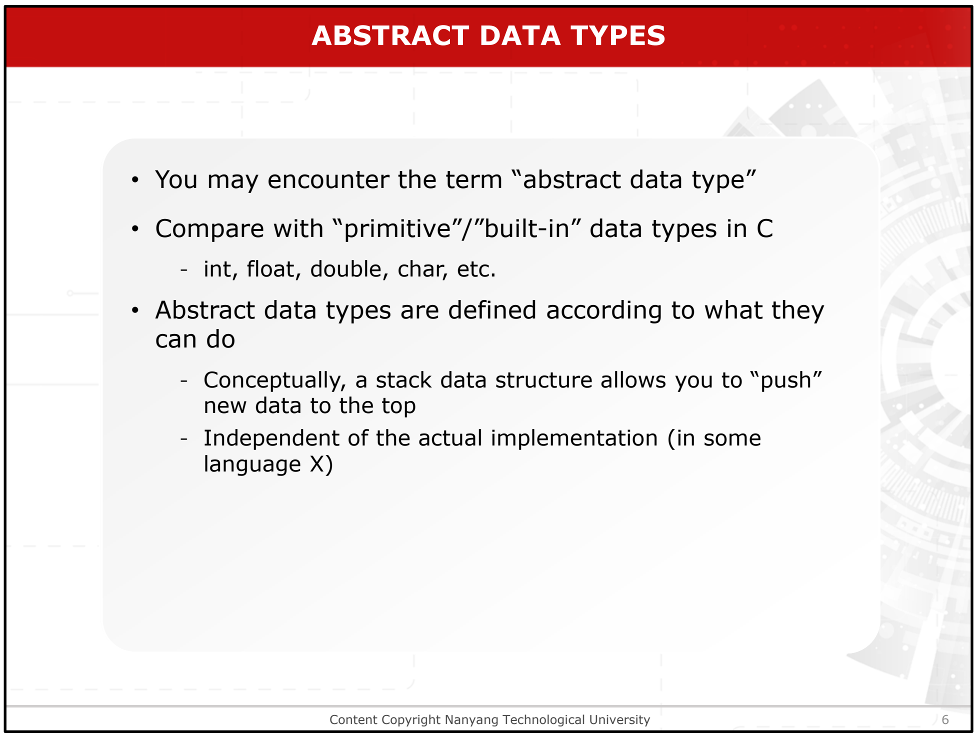
First, we will discuss what is a data structure.



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

A data structure is an arrangement of data. A specific way of data arrangement is necessary when building large scale applications on which even a microsecond difference of the run-time matters. Therefore, data arrangement is important to run the algorithms efficiently. Yet, the right way of arranging and storing your information coupled with the right algorithm to process all of that information.



ABSTRACT DATA TYPES

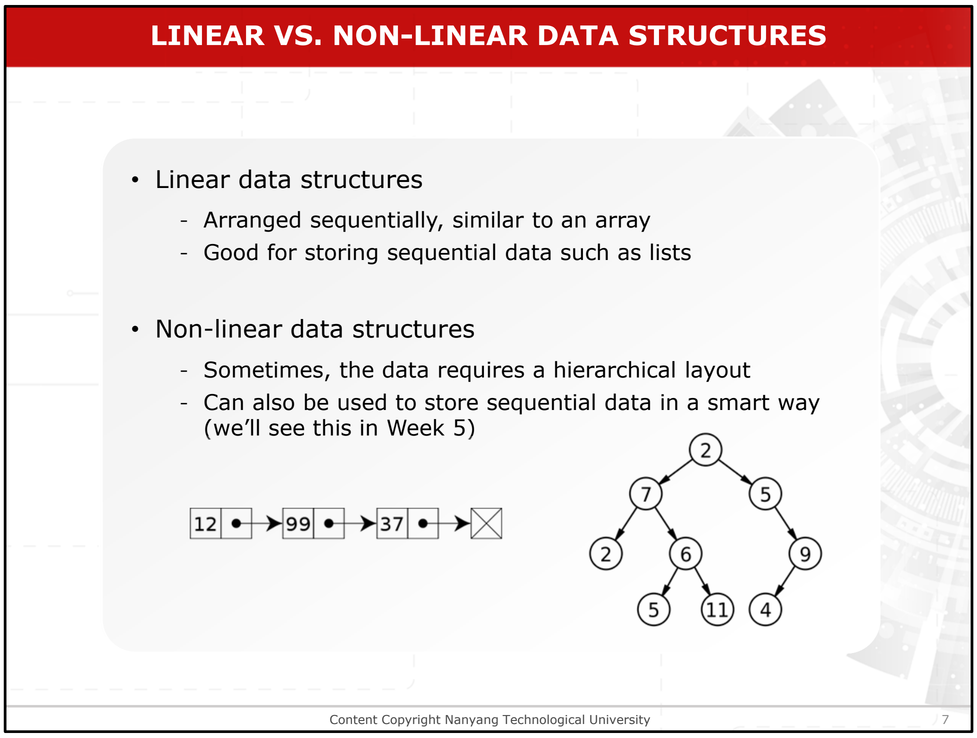
- You may encounter the term "abstract data type"
- Compare with "primitive"/"built-in" data types in C
  - int, float, double, char, etc.
- Abstract data types are defined according to what they can do
  - Conceptually, a stack data structure allows you to "push" new data to the top
  - Independent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

6

You will probably encounter the idea of an abstract data type which can also be named as an abstract data structure or dynamic data structure. You may also know its contrast with primitive or built-in data types in C such as ints, floats etc. So the idea behind all these abstract data types or what we are going to call dynamic data structures is that you have the flexibility to decide what goes into that data structure. It's entirely up to you to create a new data type beyond just integers.

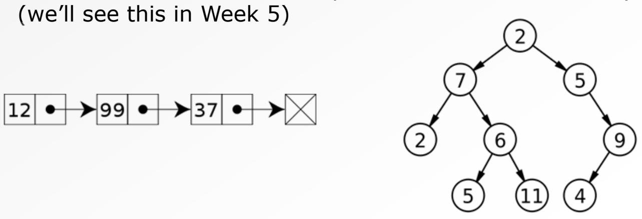
So instead of storing one number, we can create a point data structure using C struct. For example, I can wrap X and Y coordinates of a point into its own type and use for my code. By doing so, without needing to worry about X and Y coordinates separately, now I have point data type.

Some fundamental abstract data types are linked lists, stacks, queues, binary trees and binary search trees. Almost every other data structure you will encounter is either based on these fundamental data structures or is inspired by them.



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)

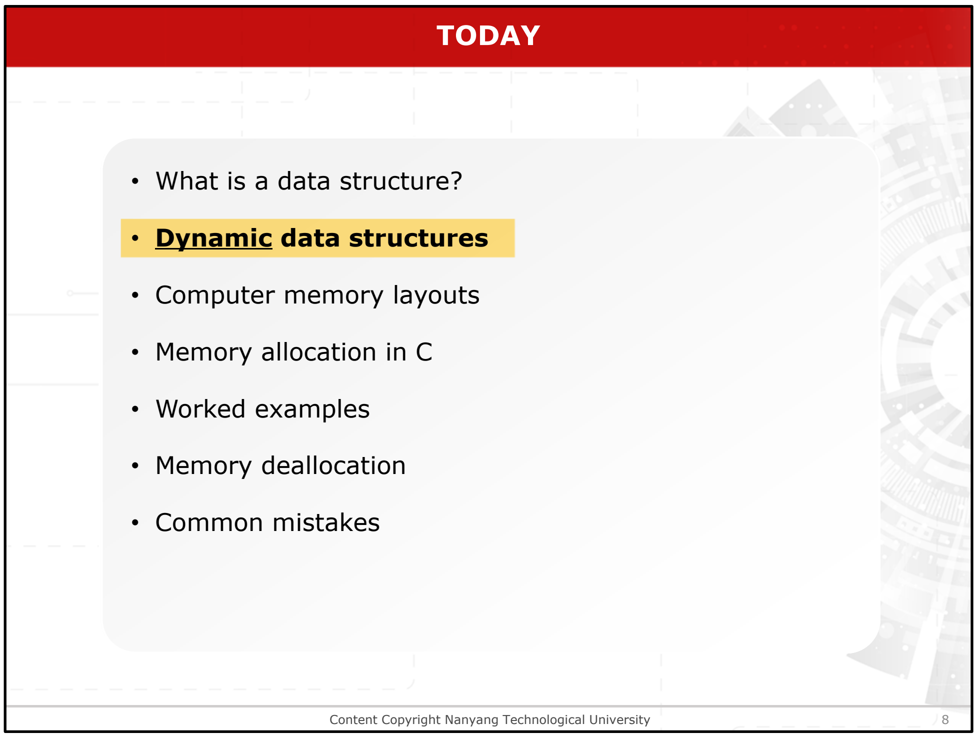


7

The difference between linear and non-linear data structures is that, in linear data structures such as linked lists, stacks and queues, you have pieces of data that are arranged in a line, one after another as in the left-side image. Once you get to your non-linear data structures, each piece of information that you have is a block of information. This block of information is not just connected to one other following block, but it connected with multiple blocks.

On the right-side image, there is a binary tree data structure. You have graph data structures where everything spreads outwards forever. There is a special arrangement of this binary tree data structure. You have trees where each of the nodes is not connected to just two other pieces, but an infinite number of pieces.

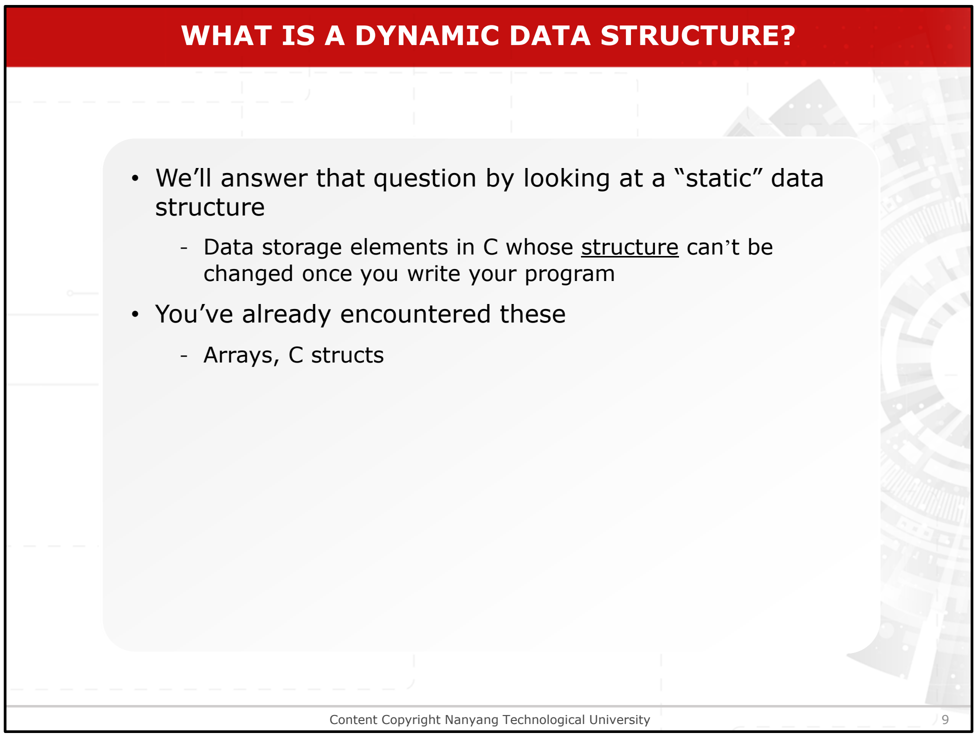
We will stick to trees where each piece of data is connected to just at most two other chunks of data. But even then, it's going to be difficult; there's going to be quite a learning curve.



TODAY

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

Now, let’s discuss about dynamic data structures.

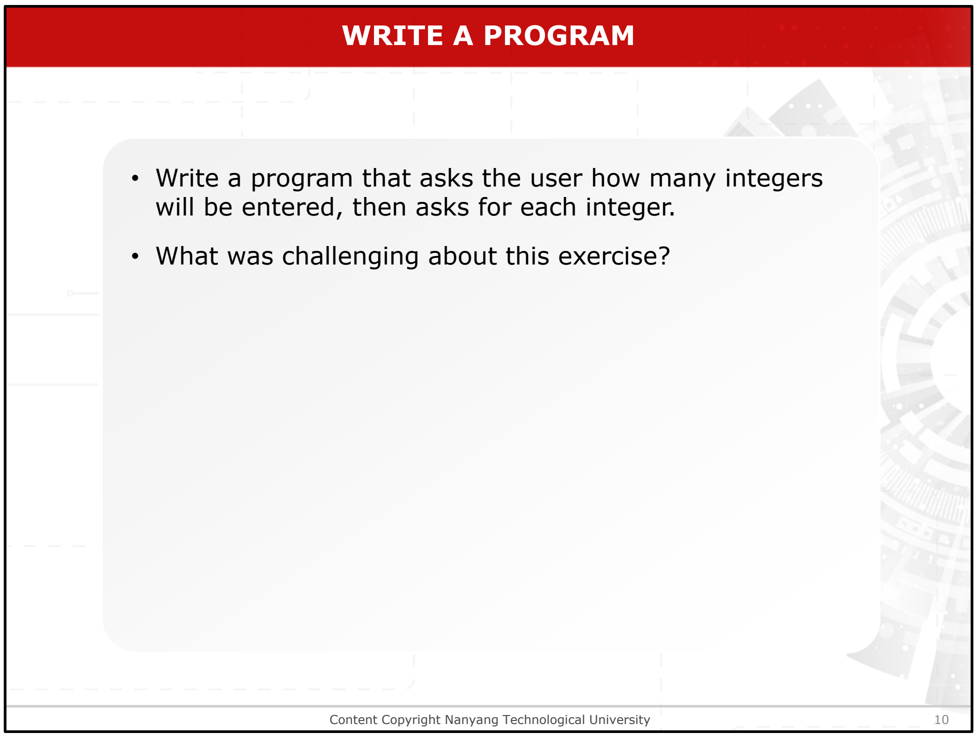


WHAT IS A DYNAMIC DATA STRUCTURE?

- 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

9

To understand a dynamic data structure, it is better to understand what a static data structure is. A static data structure is a sort of a variable that cannot change once you have compiled and run your program. Even in structs, once you have declared upfront the struct name and the member variables inside, and compiled, you will not be able to change the program. Therefore, in static data structures, we have to specify everything before we compile.

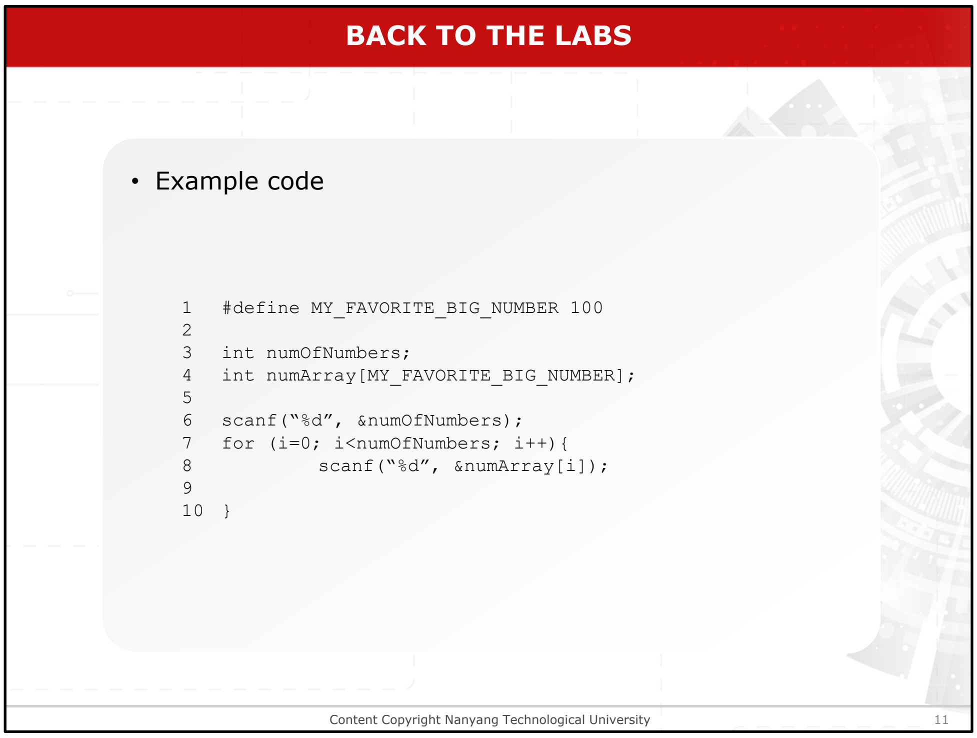


WRITE A PROGRAM

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

10

Let’s discuss about the lab 4 questions.



BACK TO THE LABS

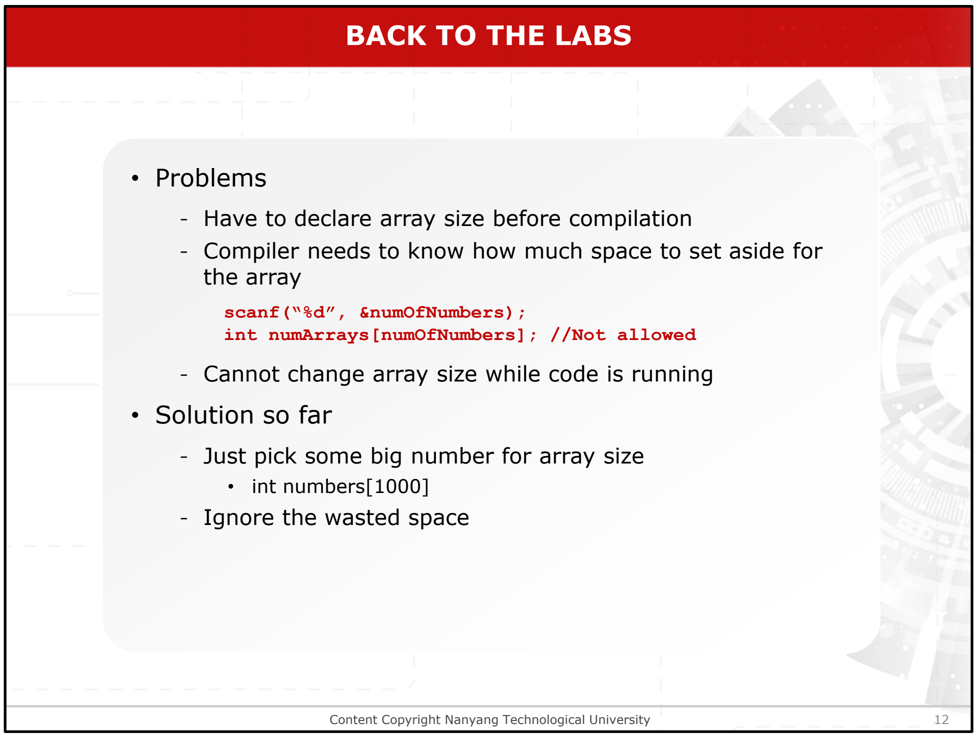
- Example code

```
#define MY_FAVORITE_BIG_NUMBER 100

int numOfNumbers;
int numArray[MY_FAVORITE_BIG_NUMBER];

scanf("%d", &numOfNumbers);
for (i=0; i<numOfNumbers; i++){
        scanf("%d", &numArray[i]);

}
```

11

The main problem that you would encounter when trying this lab question is that you would not be able to predict how many integers the user will enter. Will it be 5, 50 or 5 million?

This is a problem because you have to declare the size of your array upfront. If you correctly declared the size of the array, it will be big enough to store all the integers and program will run smoothly. But, if the array size if smaller than the number of integers that the user is going to enter, the program will overflow the bounds of the array which is bad.



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
```

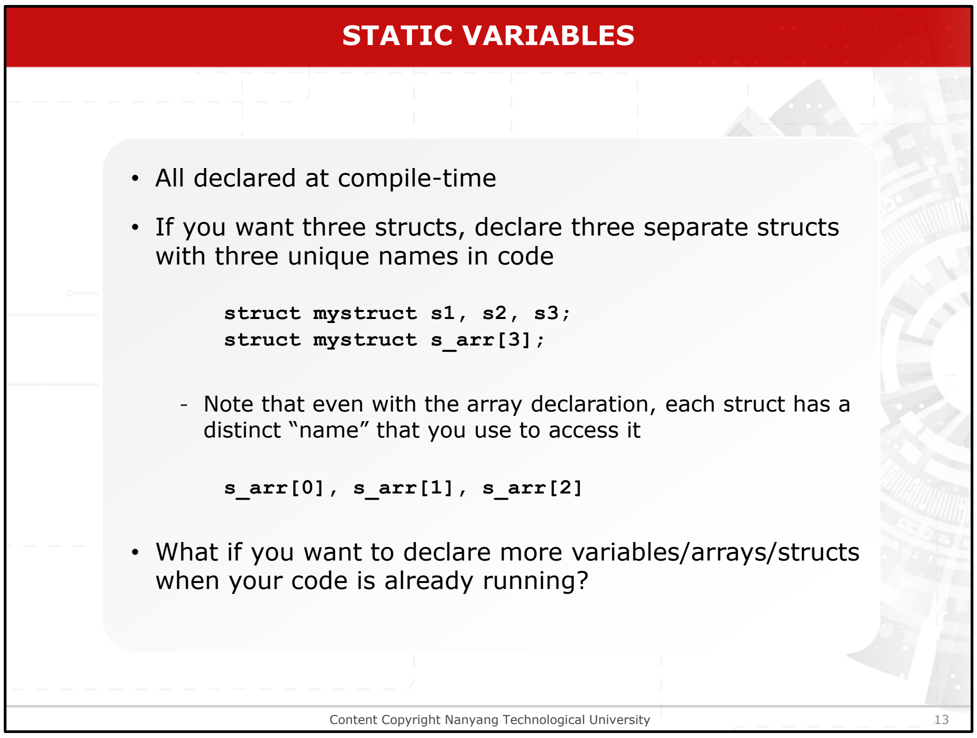
  - Cannot change array size while code is running
- Solution so far
  - Just pick some big number for array size
    - int numbers[1000]
  - Ignore the wasted space

Content Copyright Nanyang Technological University 12

The array declaration is needed before the compilation because the compiler needs to know how much memory space is needed to be set aside for every variable you have before it starts compiling. If I declare int numArray[100], the memory will set aside space for 100 integer elements which are 400bytes (4bytes for each integer). After compilation, I cannot change the array size.

Therefore, the solution to this problem is to declare the array with a bigger number. The maximum size for an integer is 2^{32} and forget about the wasted space.

Yet, this cannot be considered as a permanent solution since the memory wastage cannot be ignored easily.



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

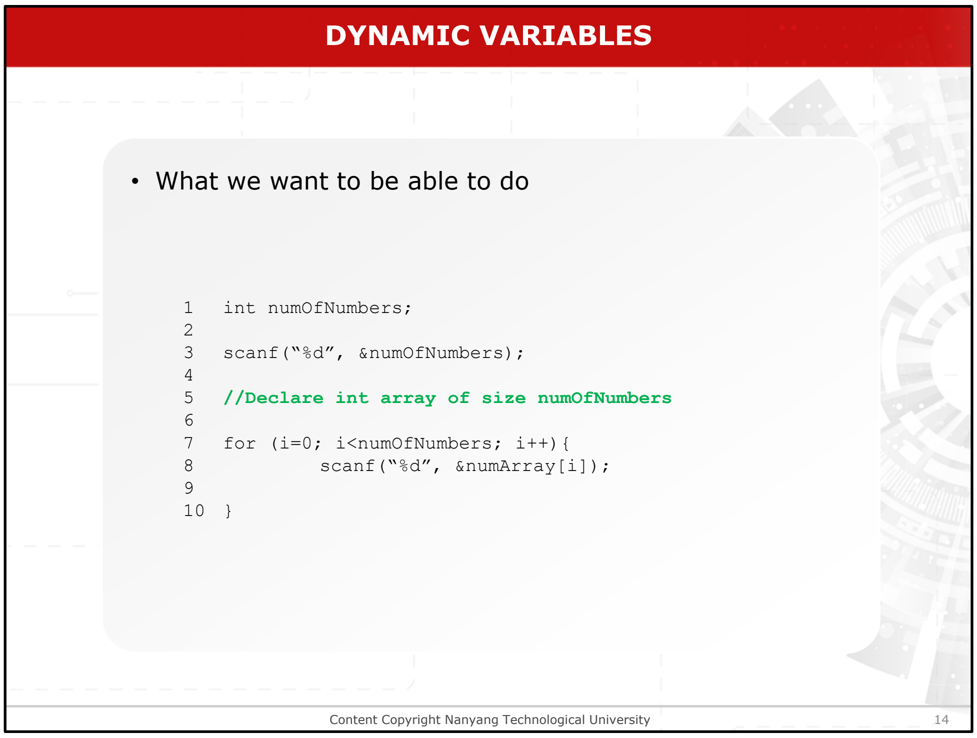
```
s_arr[0], s_arr[1], s_arr[2]
```

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

13

Structs are very important because now you have control over your data types. For example, if you have X, Y and Z coordinates inside a 3D point, you can declare 3D point data type which has X, Y and Z values inside the struct. Yet, we still have that problem on how to declare the struct?

Here, I have named 3 structs as s1, s2 and s3. Now, what if I want an extra struct, s4? I cannot declare a struct s4 while the program is running. Therefore, I declare an array. Yet, declaring s_arr[3] would limit my program for an array with 3 structs. I can't grow that array or shrink the array. Therefore, we are back in the same problem. How do we declare more elements? How do I ask for more memory to get more variables, a bigger array, or more structs when my code is already running?



DYNAMIC VARIABLES

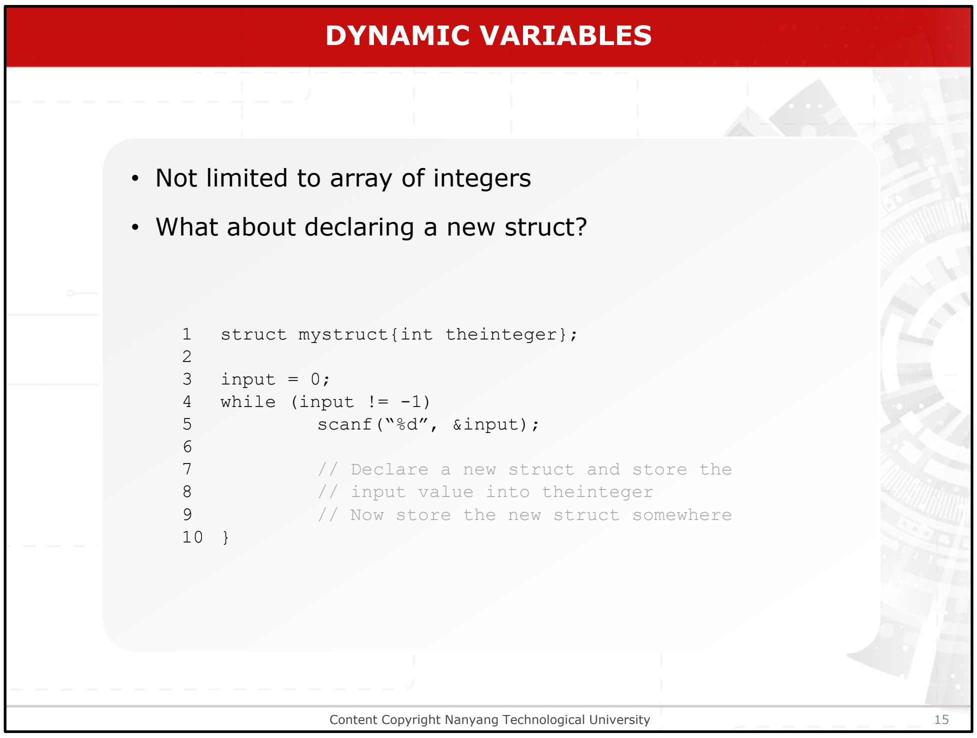
- What we want to be able to do

```
int numOfNumbers;

scanf("%d", &numOfNumbers);

//Declare int array of size numOfNumbers

for (i=0; i<numOfNumbers; i++){
        scanf("%d", &numArray[i]);

}
```

14

How to declare an int array of size numOfNumbers? The idea is that once the user enters a random number, I will inform the compiler to set an array for the entered number. The program can then allocate enough space to store the number of integers. Therefore, I will have an array of integers that is exactly the right size to hold whatever the numbers that the user is going to enter.

Now, why the wasted space matters? You may have minimum 4GB of memory in your computer, and even your phone has at least 1GB of memory. Therefore, 4bytes per integer cannot consume lots of memory. Yet, for larger industrial properly tested applications, this problem really matters. For example, Facebook cannot allocate a 1 million sized array for each user to store their friends' info just because their most popular person has 1 million friends. Instead, they allocate space for each friend when the user adds them.



DYNAMIC VARIABLES

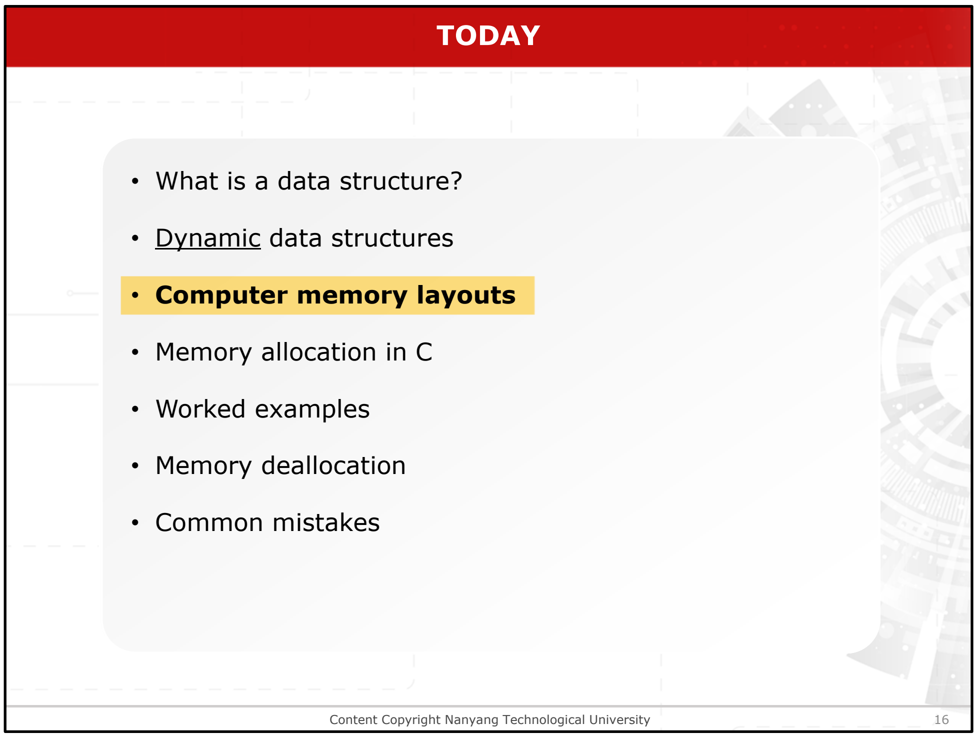
- Not limited to array of integers
- What about declaring a new struct?

```
struct mystruct{int theinteger};

input = 0;
while (input != -1)
        scanf("%d", &input);

        // Declare a new struct and store the
        // input value into theinteger
        // Now store the new struct somewhere
}
```

Content Copyright Nanyang Technological University 15

Dynamic memory allocation can be done for individual integers, individual floats, individual characters or even structs.

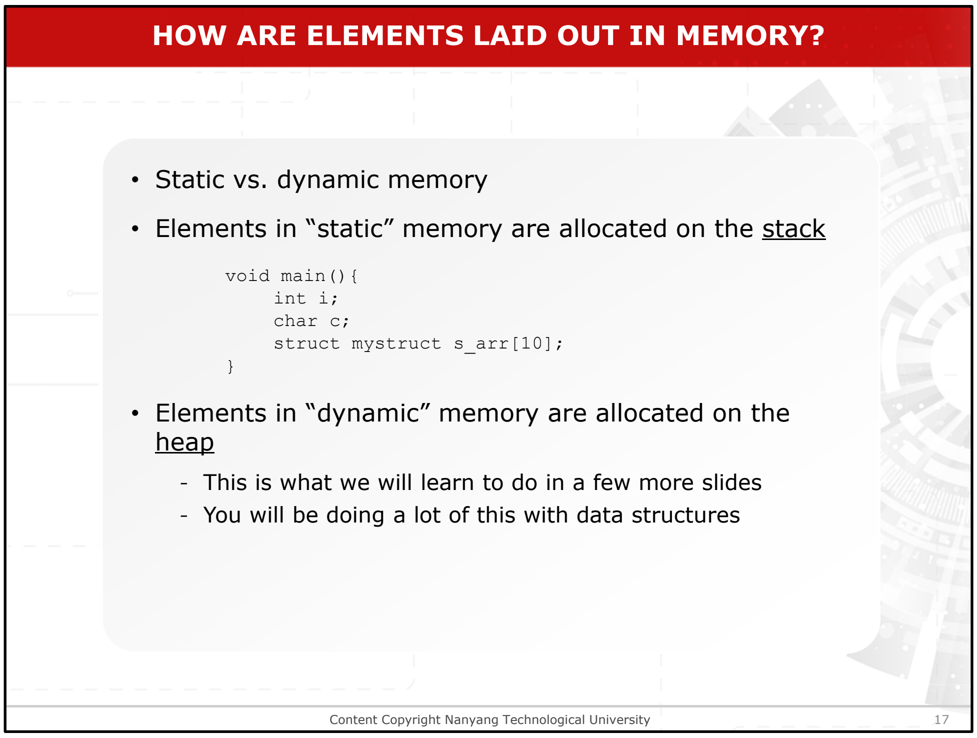
Here, **struct mystruct** has one-member field. The function has written to store values as they are input. Every time a new value has input, the function requests space for it and stores it inside the requested memory space. This process will continue until the sentinel value of -1 is input.



TODAY

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

Now we can start talking about computer memory layouts and why you have these limitations on static data structures.



HOW ARE ELEMENTS LAID OUT IN MEMORY?

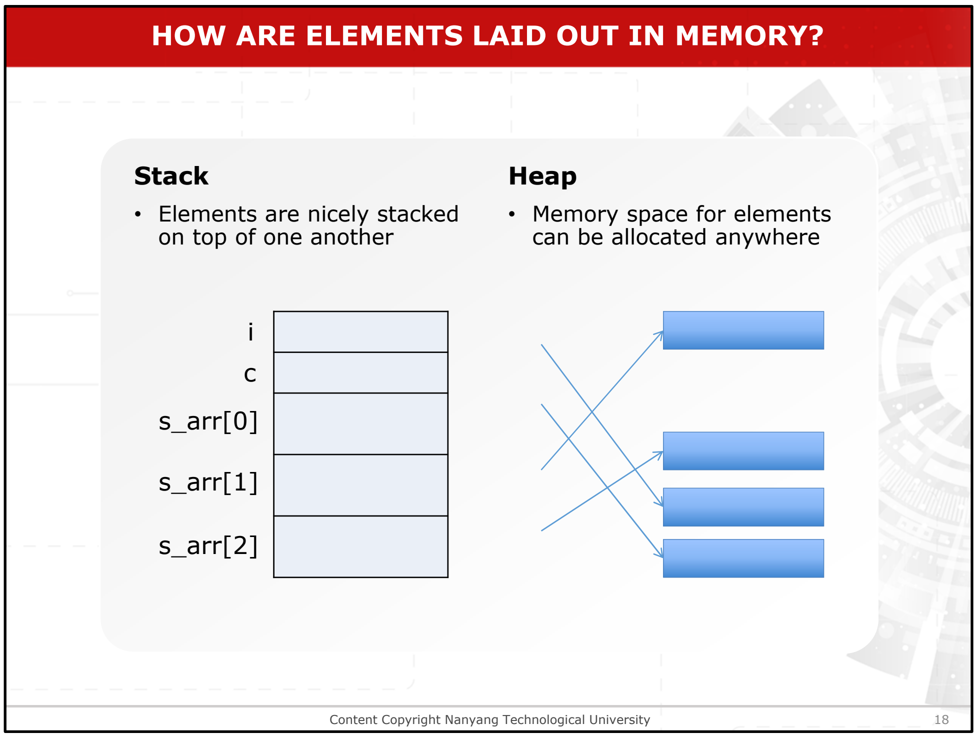
- Static vs. dynamic memory
- Elements in "static" memory are allocated on the stack

```
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

17

Now, if we have 3 variables as **integer i**, **character c**, and a **struct array of 10 structs**, these variables will be stored in the part of the computer memory named as a **stack**. Elements in the static memory are allocated on the stack. On the other hand, elements in the dynamic memory are allocated on the heap.



HOW ARE ELEMENTS LAID OUT IN MEMORY?

Stack

- Elements are nicely stacked on top of one another

Heap

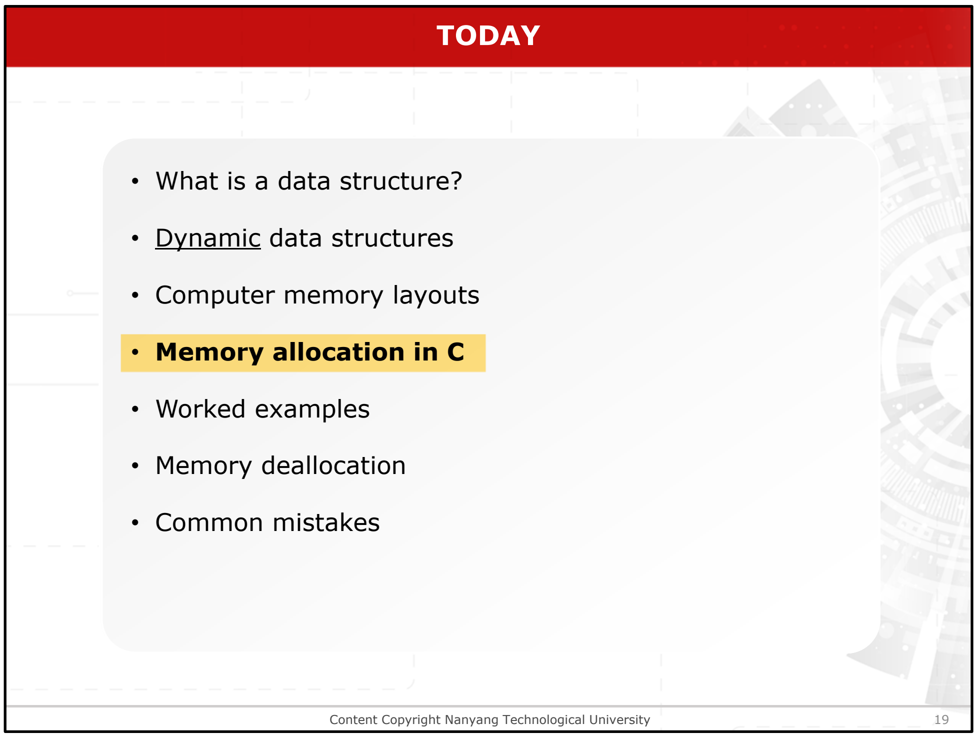
- Memory space for elements can be allocated anywhere

i
c
s_arr[0]
s_arr[1]
s_arr[2]

18

As you can see, in the stack, the declared variables are stored in the respective order contiguously. For integer i, the memory allocates 4bytes memory block. For character c, though it needs only 1byte, there is a fair chance the stack will allocate 4bytes and give access to only 1byte, because the computer memory tries to optimise certain things.

In heap, you have no idea from where the memory has been allocated. Therefore, for each request that you make, the memory will be allocated from a randomly selected block.

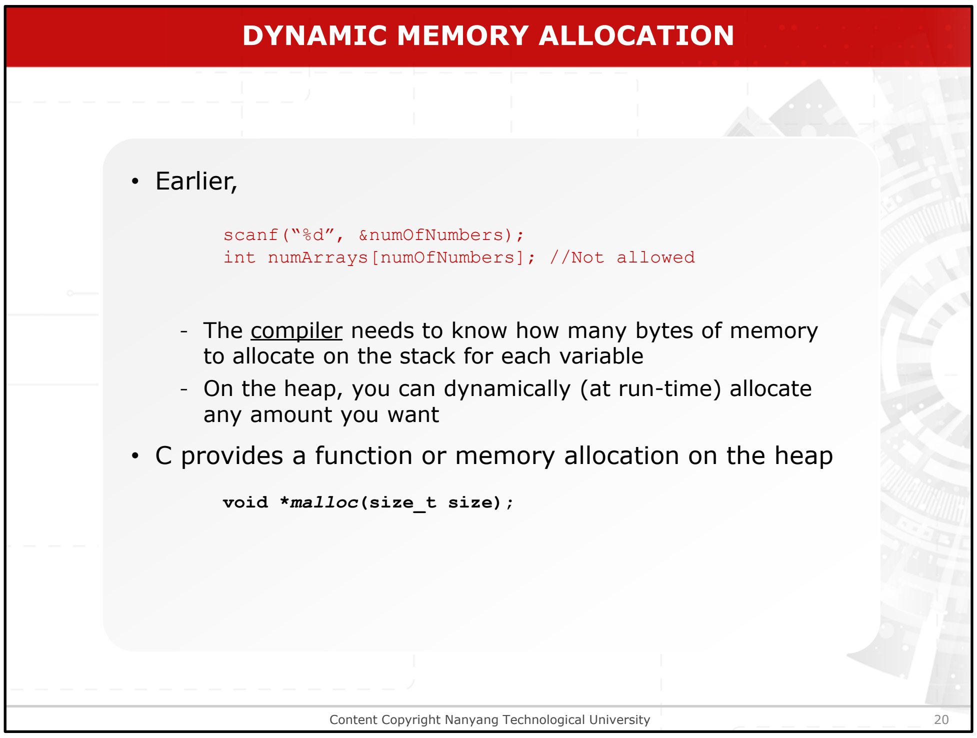


TODAY

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

19

Now, let’s learn about memory allocation in C.



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

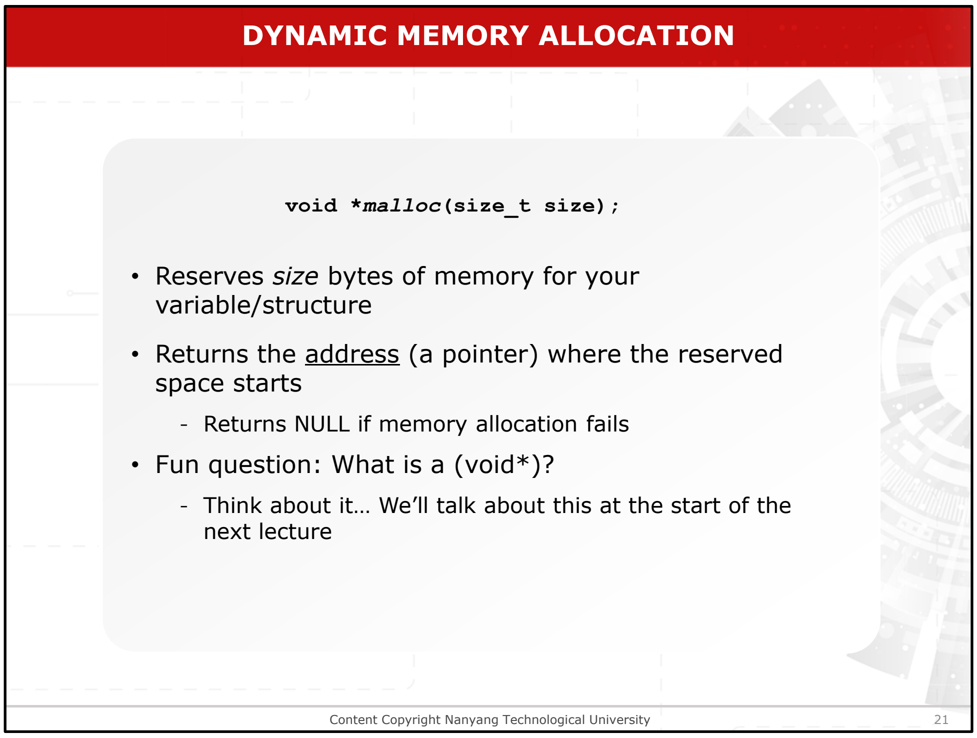
```
void *malloc(size_t size);
```

20

In C, the variable declaration should be done at the top of your code. For an example, you cannot pass for(int i=0; i++) in your for loop. You have to declare i at the beginning of your code.
Therefore, to allocate memory in heap only when needed, C introduce the function named as ‘**malloc**’. ‘malloc' is the short form of memory allocation or memory allocator.

When the malloc function is called with the required size, it returns the allocated memory address on the heap. If the requested size of the memory is not available on the heap, the function returns a NULL value.

As you can see, the return type for malloc is void *. We will talk about this type later.



DYNAMIC MEMORY ALLOCATION

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

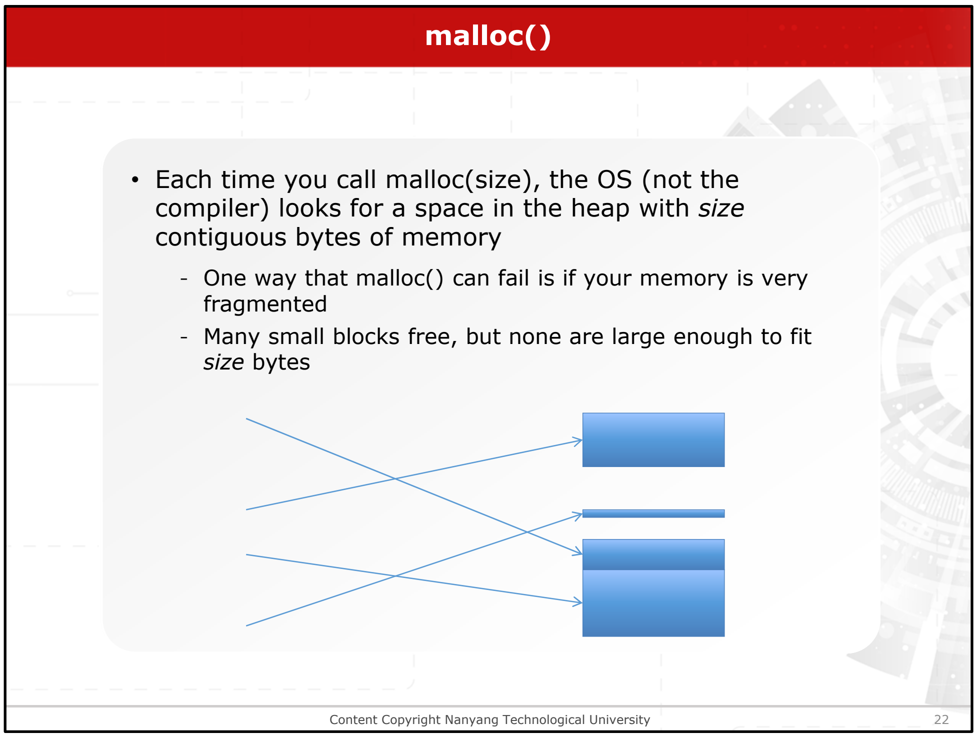
- Reserves *size* bytes of memory for your variable/structure
- Returns the address (a pointer) where the reserved space starts
  - Returns NULL if memory allocation fails
- Fun question: What is a (void*)?
  - Think about it… We'll talk about this at the start of the next lecture

21

In C, the variable declaration should be done at the top of your code. For an example, you cannot pass for(int i=0; i++) in your for loop. You have to declare i at the beginning of your code.
Therefore, to allocate memory in heap only when needed, C introduce the function named as '**malloc**'. 'malloc' is the short form of memory allocation or memory allocator.

When the malloc function is called with the required size, it returns the allocated memory address on the heap. If the requested size of the memory is not available on the heap, the function returns a NULL value.

As you can see, the return type for malloc is void *. We will talk about this type later.

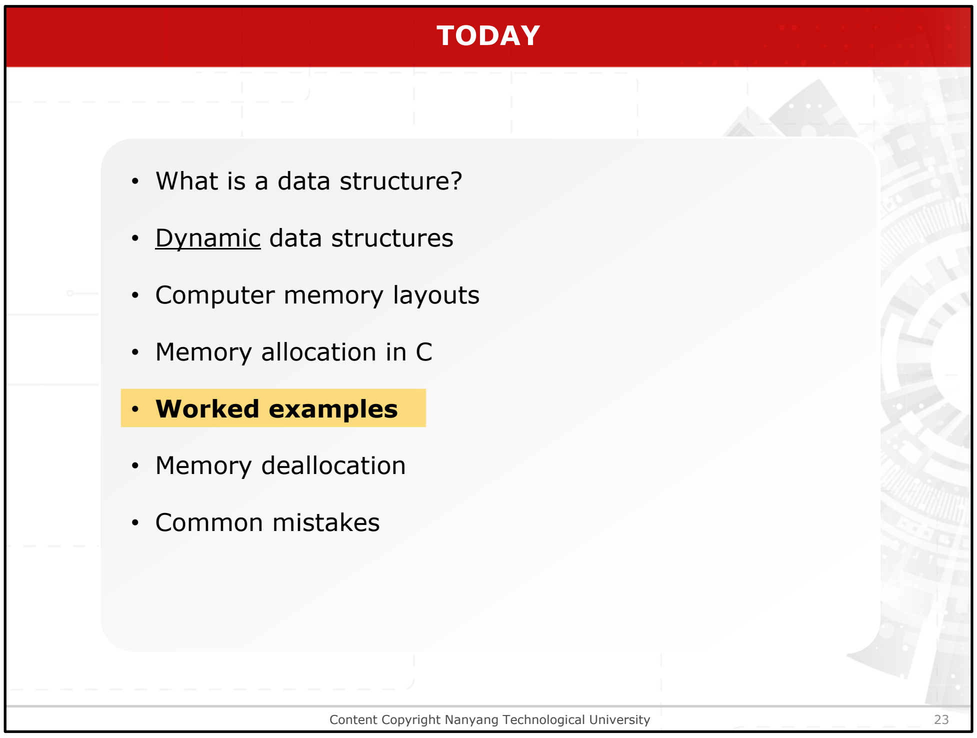


malloc()

- 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

Why would malloc fail?

When we request memory from the heap, it grabs whichever free space that is available to which the compiler goes first. The compiler has its own algorithm which works together with the IS to decide. Let's say I need 20bytes. The compiler goes to the address 1056 and gives you 20 bytes. Eventually, the heap is filled with blocks that have been allocated for various malloc calls. Therefore, if I have 1GB memory and 750MB is already allocated for different programs, the heap will be failed to serve a request of 500MB. If we consider another scenario and request 200MB which is available in the heap but in non-contiguous memory chunks as in 120MB memory block and an 80MB memory block, the malloc is going to be failed because the memory chunks are fragmented.

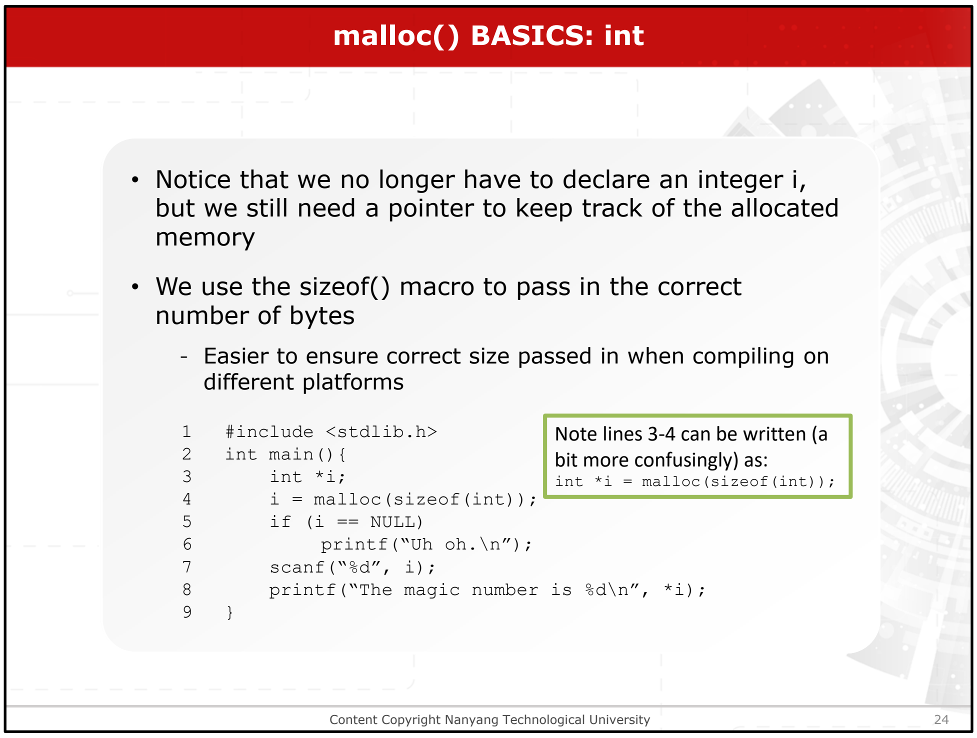


TODAY

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

23

A few examples of how to use malloc to allocate space for a single integer, for an array, and possibly for a struct if necessary.



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   }
```

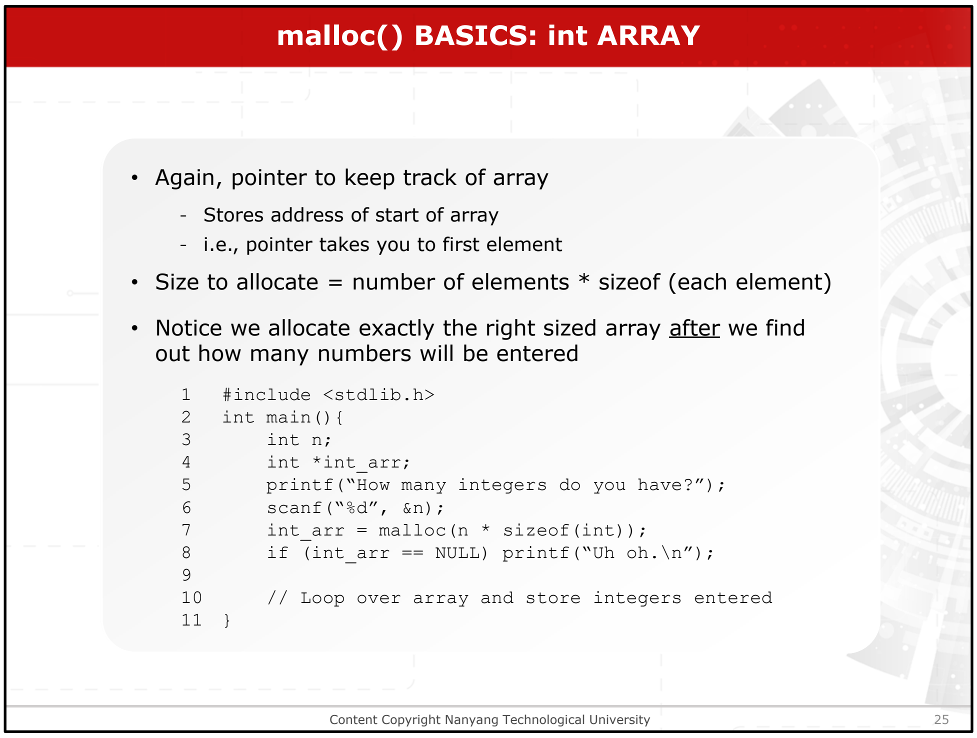
Note lines 3-4 can be written (a bit more confusingly) as:
`int *i = malloc(sizeof(int));`

24

How to allocate memory for an integer using malloc?
Here, you can see that instead of declaring integer i, we declare a pointer to keep track of the allocated memory. Then, malloc needs to pass a number of bytes that are needed from the heap. For an integer, we need 4bytes from the heap. Then why we pass **sizeof(int),** instead of 4bytes? Because, on different platforms such as mobile or embedded board, the size of an integer is less than 4bytes. Therefore, instead of passing the number (4bytes) we pass **sizeof(int)** so that the compiler allocates the correct value based on the platform.

Once the malloc function has been called, it allocates space for an integer and brings the address of the allocated memory block. Now we need to store that address because without that address we cannot reach to the allocated memory. Since we are going to store a memory address, we use a pointer (*i).

There is a bit more confusing way of writing this code chunk. You can see that is written inside the box. Therefore, I have broken the code into less confusing lines.
Line 3 – declaring integer *i
Line 4 – get the allocated memory address and assign it to i
Line 5 & 6 – for sanity check just in case if malloc fails to perform
Line 7 – scanf to pass i instead of ampersand i to input the address of the variable to go directly to the variable in the memory and change the value



malloc() BASICS: int ARRAY

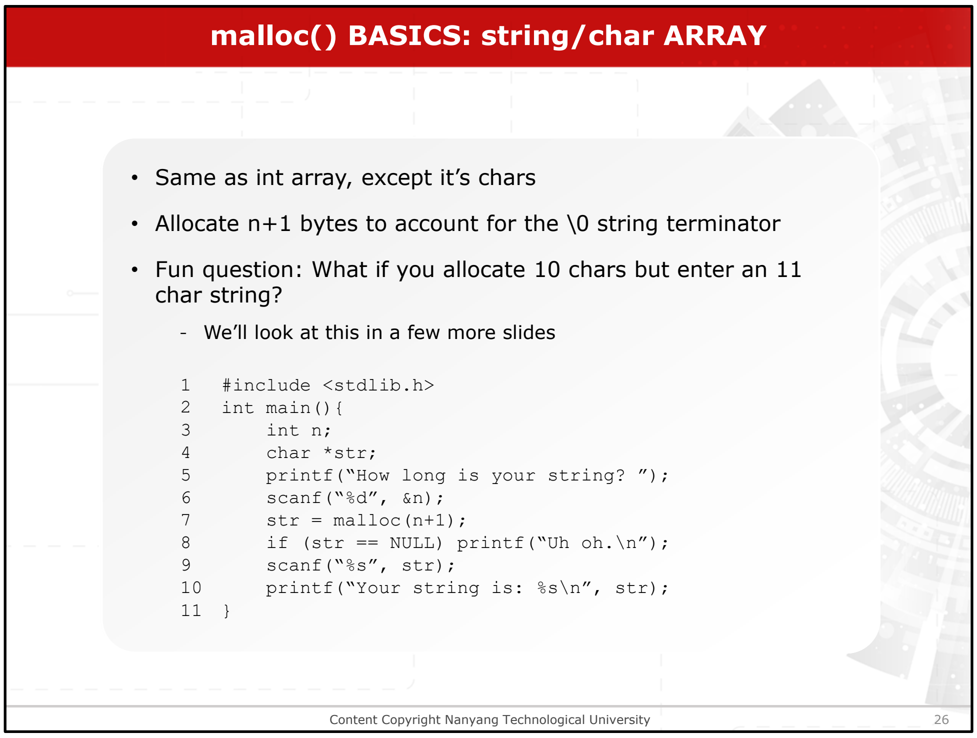
- Again, pointer to keep track of array
  - Stores address of start of array
  - i.e., pointer takes you to first element
- Size to allocate = number of elements * sizeof (each element)
- Notice we allocate exactly the right sized array after we find out how many numbers will be entered

```
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  }
```

25

When we allocate memory for an array of 10 integers, we need to pass 10*4bytes or 10*sizeof(int) because now we need space to store 10 integers. Therefore, as in Line 7, we pass **n*sizeof(int)** for n number of integers.

Here, we declared int_array as a pointer (*int_array). We have to do this because the malloc returns with the memory address which has to store as a pointer. Yet, in C, arrays and pointers are interchangeable in a certain situation. Here, we are treating the *int_array as an array though it has just the starting address. Now, since I have the pointer to the starting address of the integer array and I have declared int_array as int *int_array, to get to the address of the next element of the array, I should move down 1 integer worth of bytes which is 4bytes. By moving down 4bytes each time, I can reach all the elements in the array.

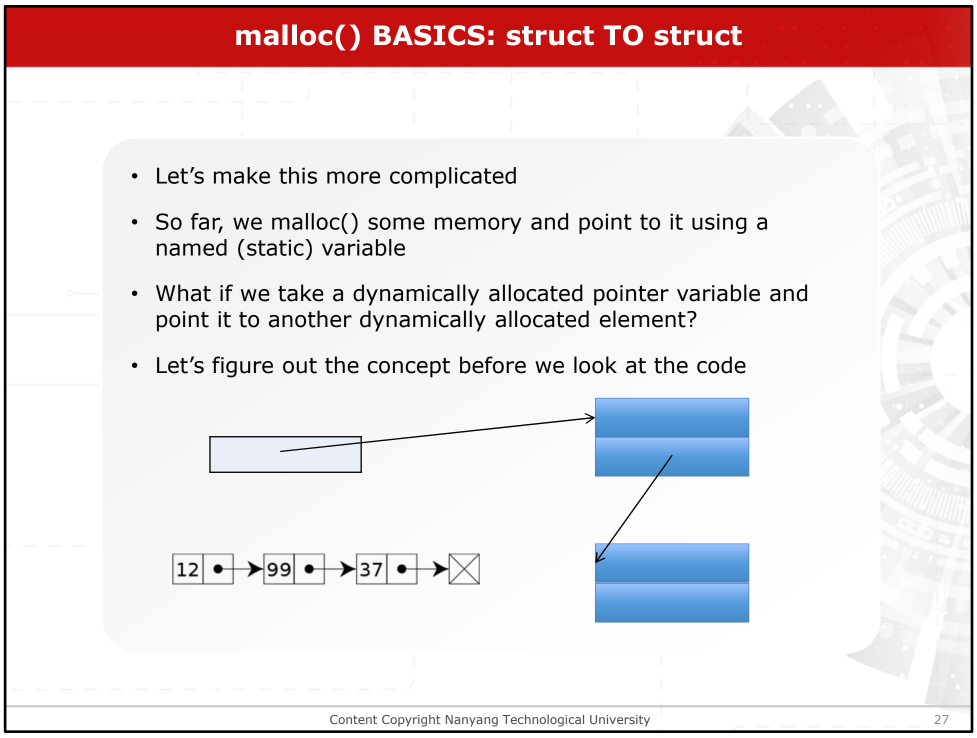


malloc() BASICS: string/char ARRAY

- 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

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

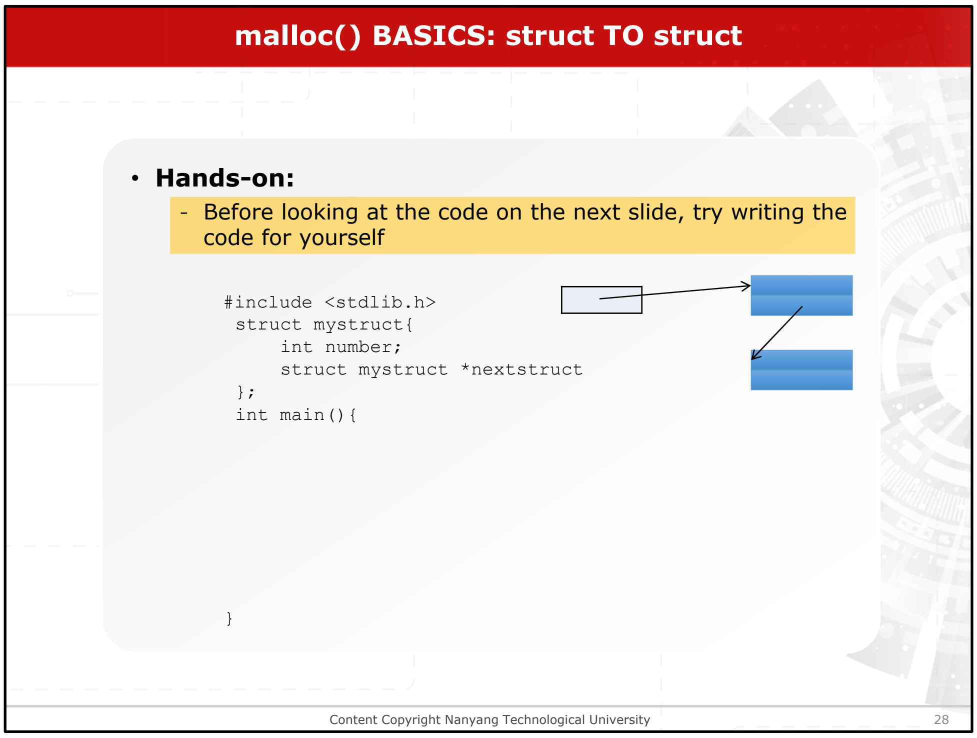
The same process goes with string or character arrays. To allocate enough memory space for the string, we pass n+1 for malloc where n is the number of characters. We do not have to pass sizeof(char) because the size of a character is 1byte. Yet, for proper and complete code, you can pass sizeof(char). We use n+1 to leave a space for the /0 at the end of the array of characters.



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

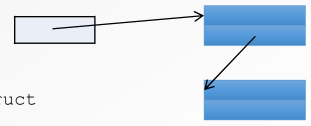
Using a pointer you can keep track of everything such as create allocate new memory blocks, new structs, link structs, delete allocated memory, etc. Now, if you look at the given picture, the first blue block is a C struct. In the struct, the part can be an integer variable. The second part is another pointer. So, I use malloc to allocate memory for the struct and the second variable of the struct points me to the second dynamically allocated struct.



malloc() BASICS: struct TO struct

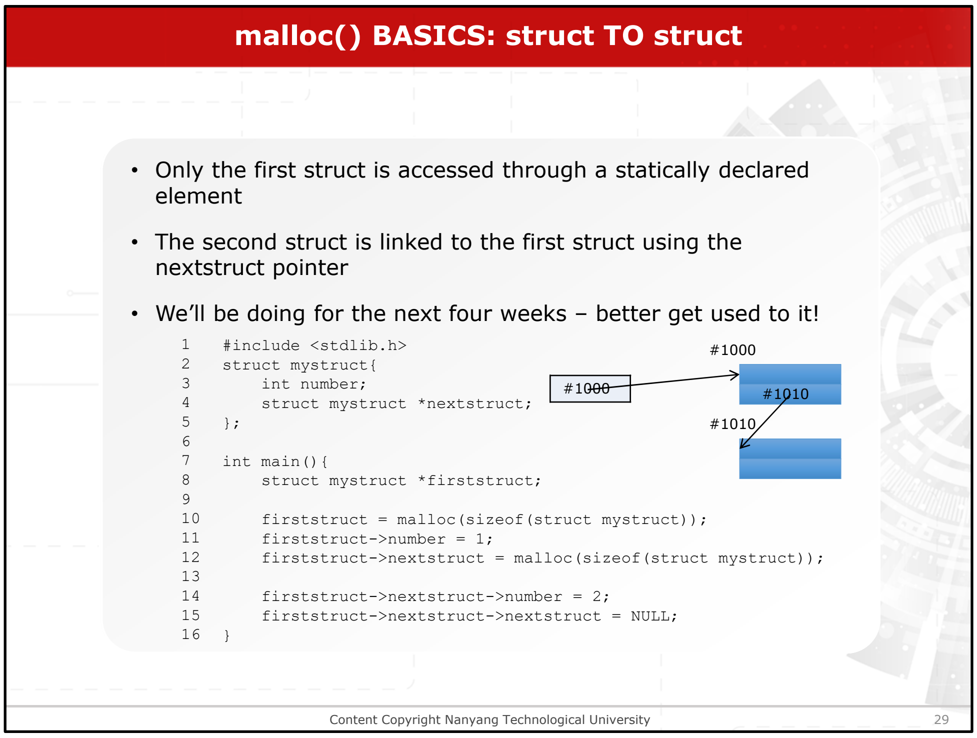
- **Hands-on:**
  - Before looking at the code on the next slide, try writing the code for yourself

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





}
```



Content Copyright Nanyang Technological University 28

The **struct mystruct** has two members inside. The first one is an integer, and the second one is a pointer which creates a link to a different chunk of data.



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   }
```

Content Copyright Nanyang Technological University 29

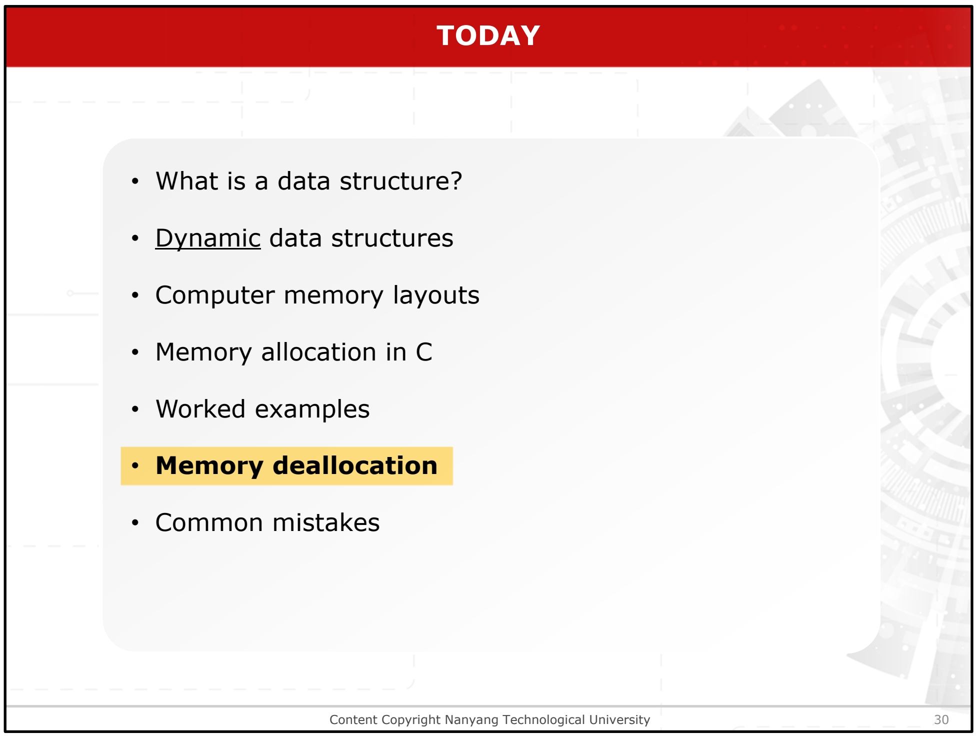
Line 3 declared integer number. This is not important since you can throw in whatever number values that you want to store inside. The more important thing is to make sure that all of these links get set correctly. You need to make sure the pointers get the correct address value stored inside those variables.

In the code, I have two pointer variables. The first pointer is **firststruct** which is the pointer to the **struct mystruct**. This pointer has been declared in line 7 is statistically allocated. Therefore, it comes from the stack portion of the memory. Therefore, once you compile your code, it knows that it needs to allocate 4bytes of memory to hold the address, the pointer variable.

After that, from line 9 onwards, everything is being dynamically created. Where the function pass **firststruct=malloc (sizeof(struct mystruct))**, it request enough memory from the heap to store that structure. Once malloc returns with the address in the heap, it is stored in the firststruct pointer variable and creates the first link. As you can see in the diagram, the firststruct is at #1000, and that value 1000 gets stored into the firststruct pointer variable.

Now we need to connect the first dynamically allocated struct to the second structure. To do so, we need to call malloc function again using the pointer variable inside the firststruct. Once the malloc returns with the address to the nextstruct which is #1010. Then, we will pass that value to the nextstruct pointer variable which is inside firststruct.

So basically, we can keep adding new structs. The basic idea here is that we can access every struct because they are all linked together by this chain of pointers. And that's a fundamental idea behind a linked list.

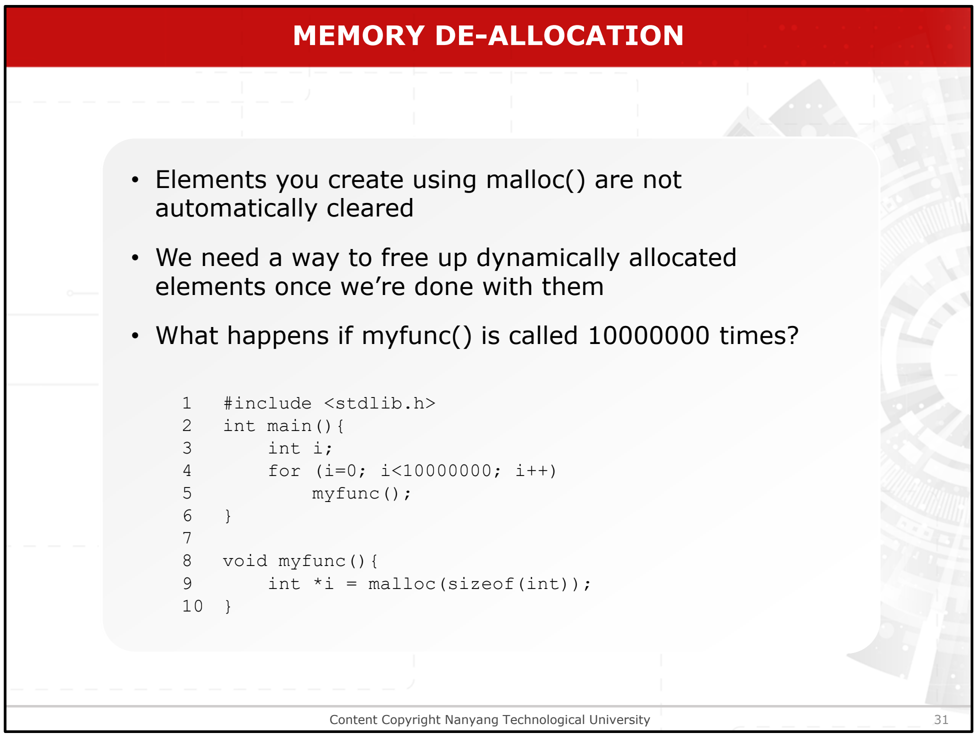


TODAY

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

30

What happens when you ask for memory, and your program runs for a long time? What happens when you no longer need that chunk of memory that you requested? You do need a way to give it back to the system. Otherwise, you are going to hold on to all this memory that you are not using anymore.



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?

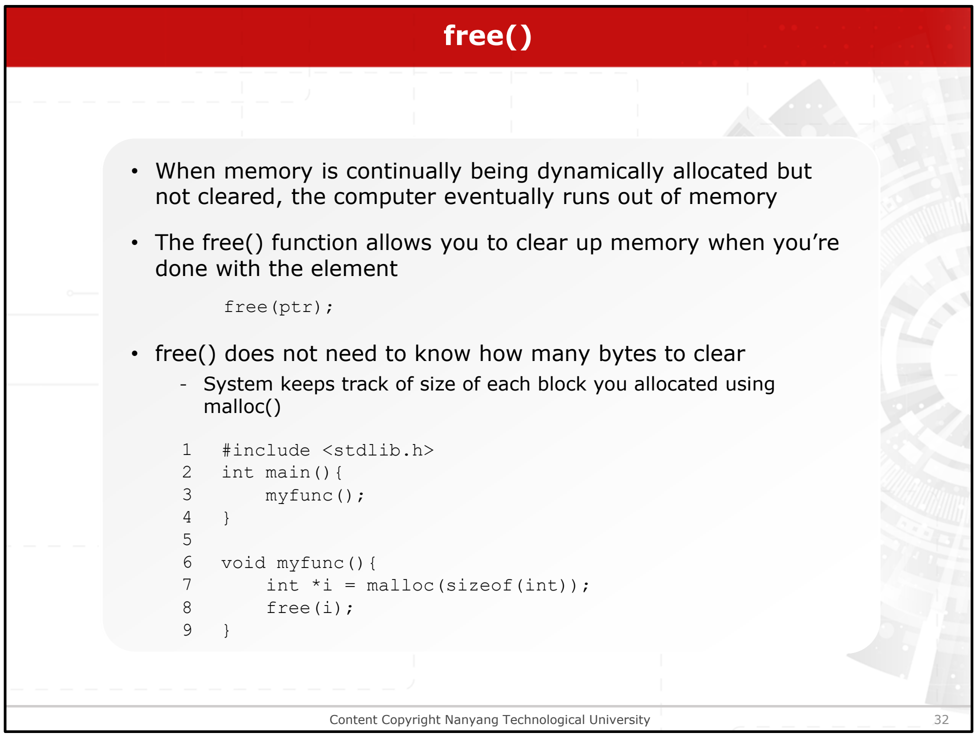
```
1   #include <stdlib.h>
2   int main(){
3       int i;
4       for (i=0; i<10000000; i++)
5           myfunc();
6   }
7
8   void myfunc(){
9       int *i = malloc(sizeof(int));
10  }
```

31

Memory de-allocation is important because, anything that you request using malloc function does not automatically get deallocated. You have to actually explicitly call a function which will return the allocated memory back.

Here, we have the main function named myfunc which declared between line 8 to 10. It runs for 10million times.

For each time we call this function, I call malloc function to allocate 4bytes of memory from the heap to store an integer variable using a pointer variable i. the scope of the pointer variable i is internal to the myfunc function. Therefore, when the function ends, i does not exist anymore. Yet, the allocated memory will be kept on allocated. This is because your system does not recognize that the allocated memory will not be used anymore. This is a problem because, if we keep on calling myfunc function, at a point there will not be enough memory in the heap.



free()

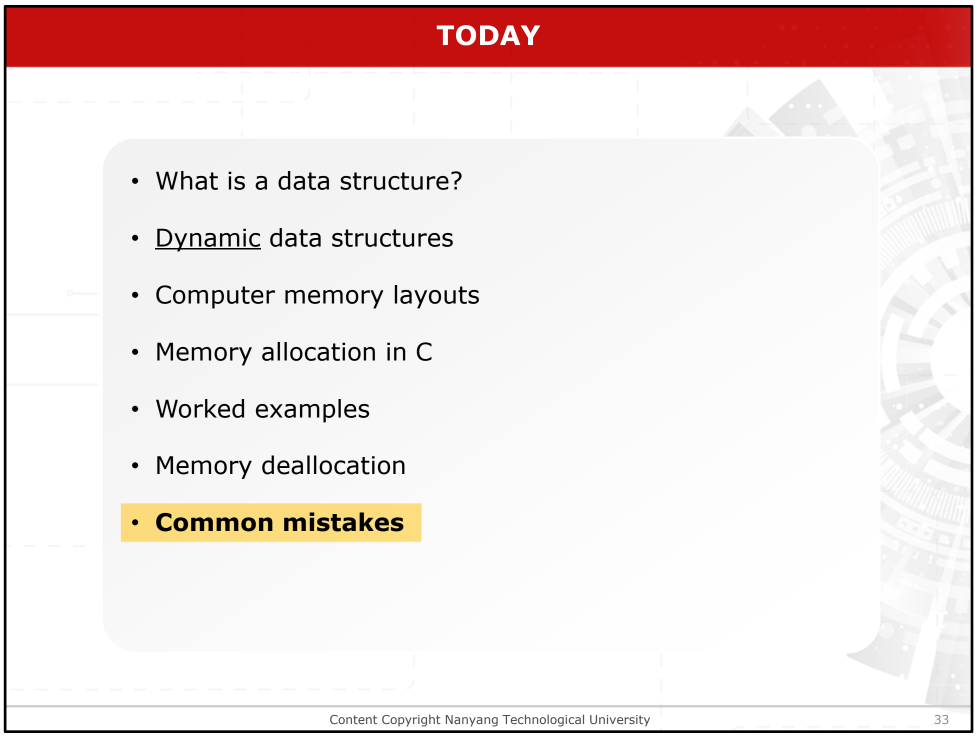
- When memory is continually being dynamically allocated but not cleared, the computer eventually runs out of memory
- The free() function allows you to clear up memory when you're done with the element

```
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()

```
#include <stdlib.h>
int main(){
    myfunc();
}

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

In malloc, you request a certain number of bytes, and it returns with the address. For free, you pass the address given by malloc and request to clear the memory for you. You do not have to pass the number of bytes that need to be cleared because your OS keeps track of those data.

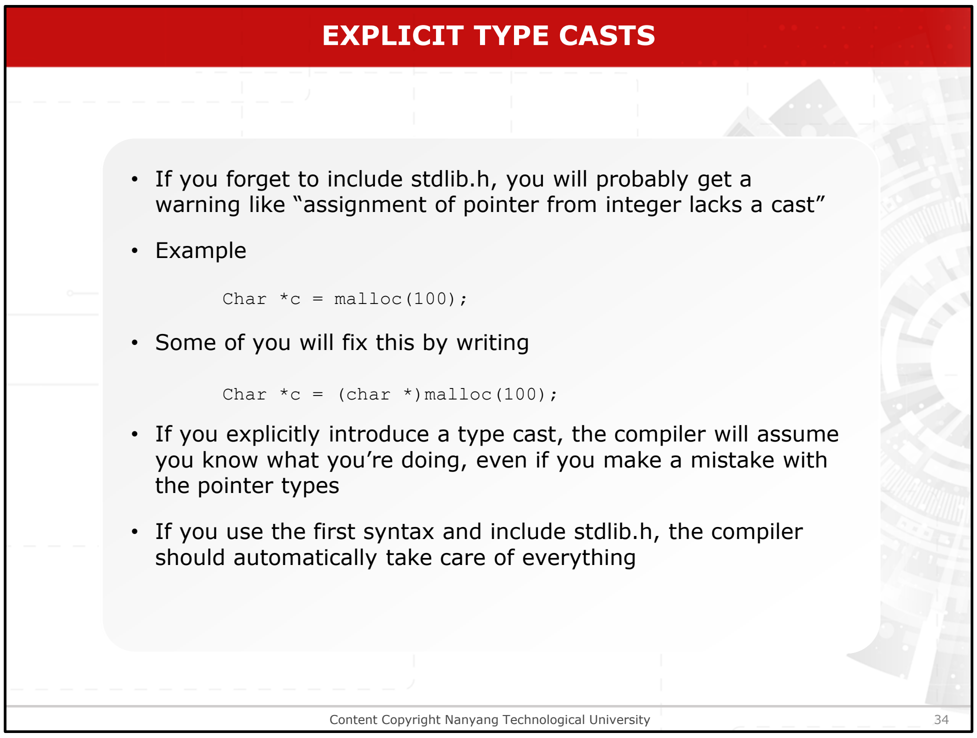


TODAY

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

33

Now, let’s see what the common mistakes we make.



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);
```

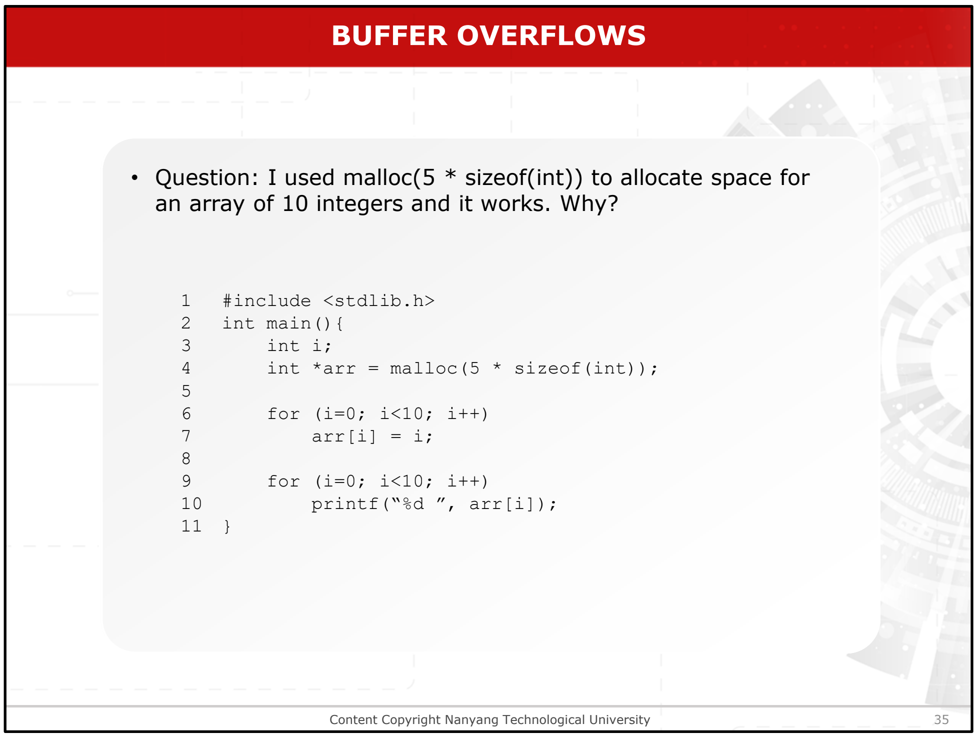
- Some of you will fix this by writing

```
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

34

Never forget to include stdlib.h if you are writing standard C code. Visual studio sometimes compiles your code as a C++ code, which would not cause a problem except that in C++ standard, you need to set the correct data type of the pointer or the address that returns with malloc. For example, if you want to store the address in a character pointer, you need to cast it. To avoid this, you should include stdlib.h.



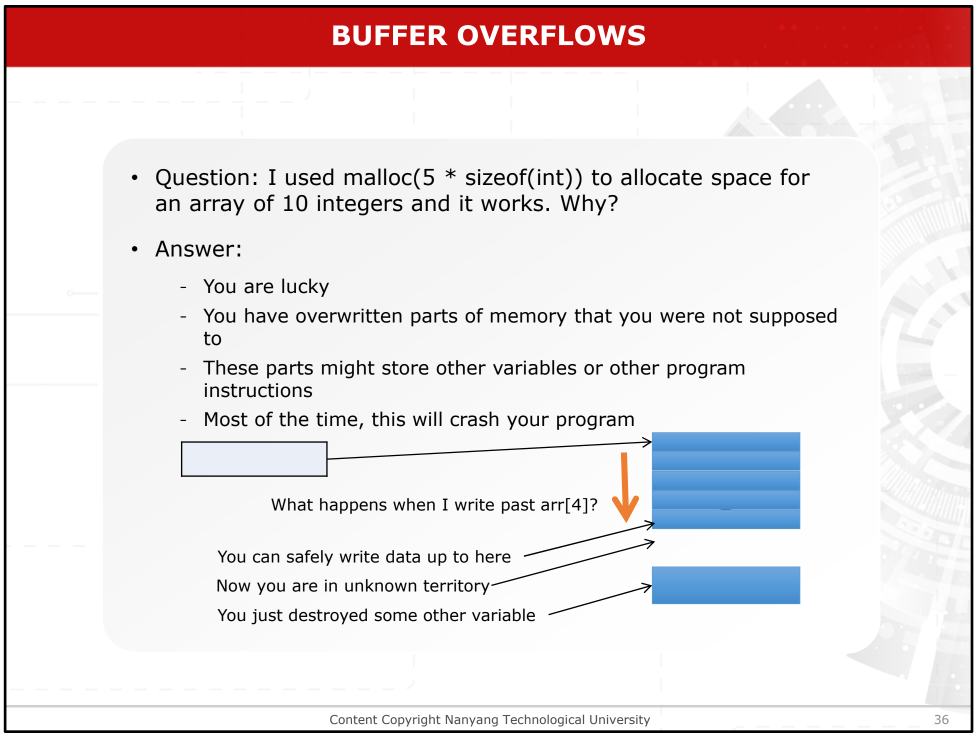
BUFFER OVERFLOWS

- 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  }
```

Now, this question is quite interesting because it tests you on your understanding of arrays, how much space you have, and what happens when you start to overflow the bounds of an array. As you have learnt, when I pass malloc(5 * sizeof(int)), you know that malloc will allocate enough space for an array of five integers.

Then, by using line 6 and 7, I am trying to store 10 integers, and in line 9 and 10, I am trying to print the numbers I have assigned. What will this piece of code print?



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

What happens when I write past arr[4]?

You can safely write data up to here

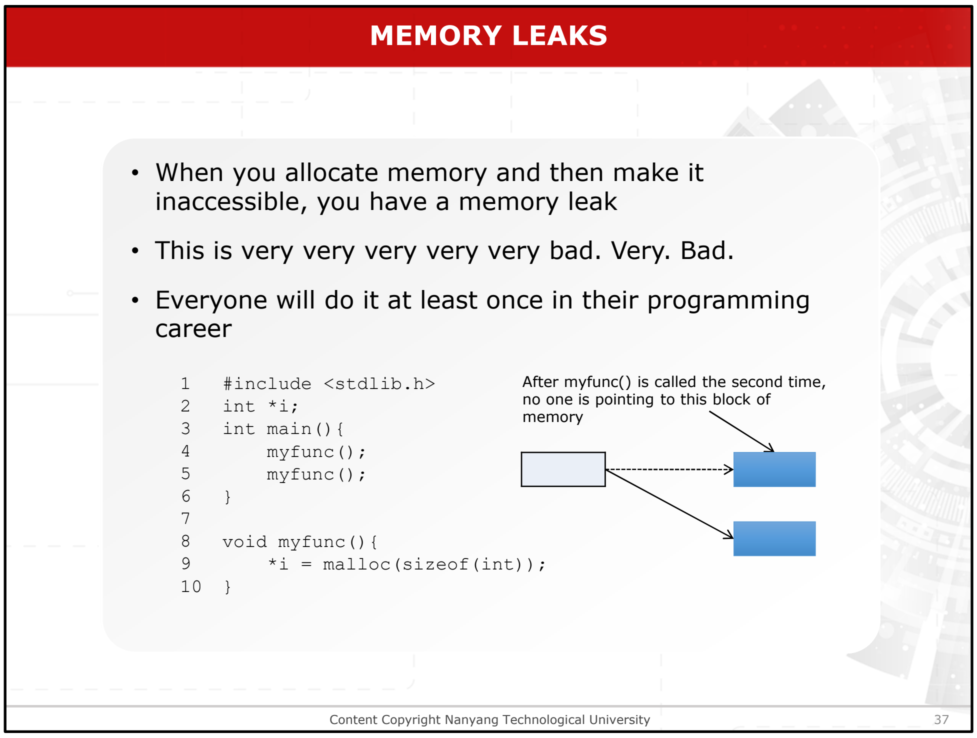
Now you are in unknown territory

You just destroyed some other variable

Content Copyright Nanyang Technological University 36

And if you look at the picture over here, we are looking at the heap on the right. The declared pointer variable *arr is the on the left. And I can see that I have five spaces, each one representing four bytes, one integer’s worth of memory, that I can write integers into.

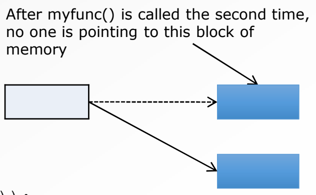
Now, as I'm writing into index 0, 1, 2, 3, 4, I'm writing into space that actually has been reserved for me. But what happens when I start going to array 5, 6, 7, 8, 9?
I will write the integers after array index 4 on memory that is allocated to some other program. Those can be important system variables which can crash the system.



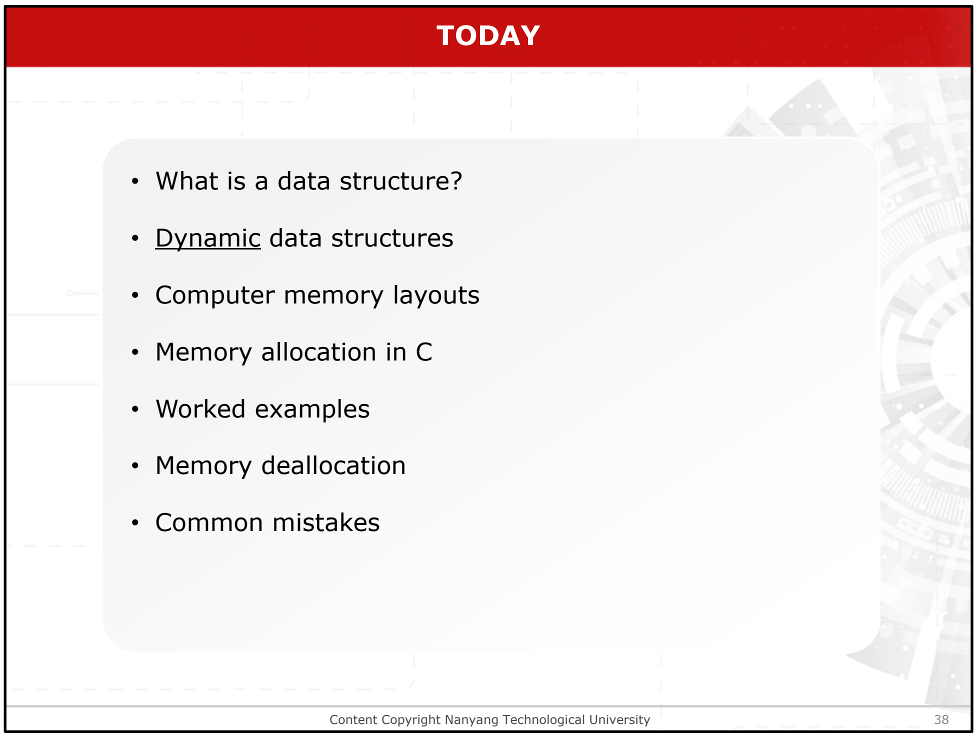
MEMORY LEAKS

- 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

```
#include <stdlib.h>
int *i;
int main(){
    myfunc();
    myfunc();
}

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


After myfunc() is called the second time, no one is pointing to this block of memory

When you call malloc function and forget to call free, that allocated memory block will continuously allocate to you. Once you finished the function, you will lose the address of the memory block as well. This is called memory leak and it will affect the system badly.

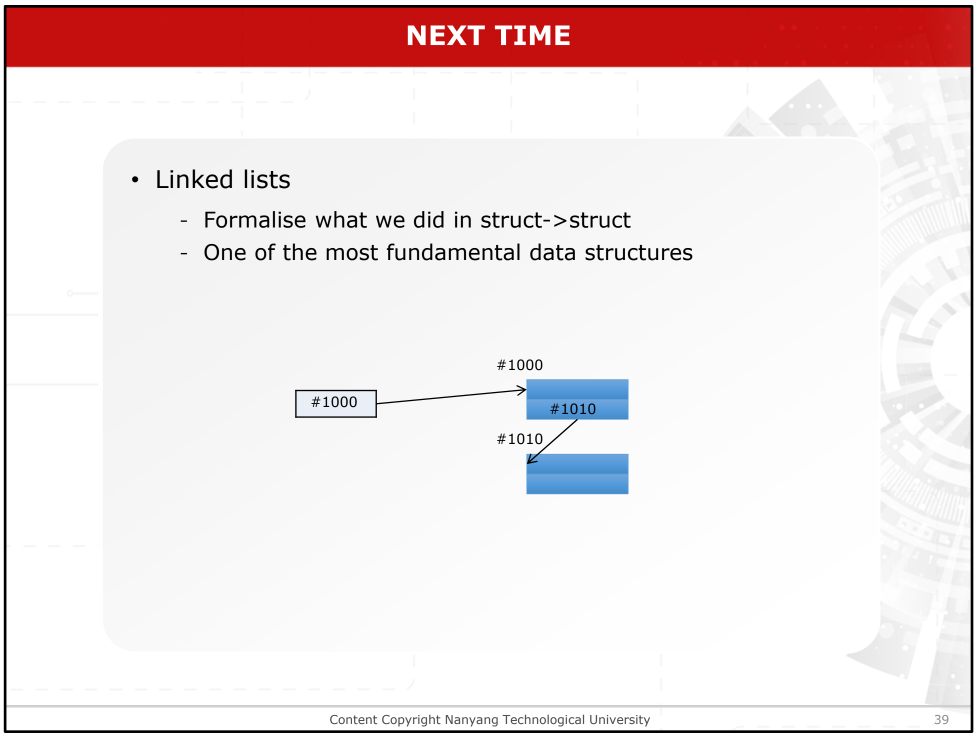


TODAY

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

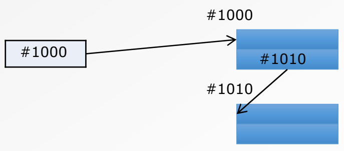
38

We have covered these topics in today’s lecture.

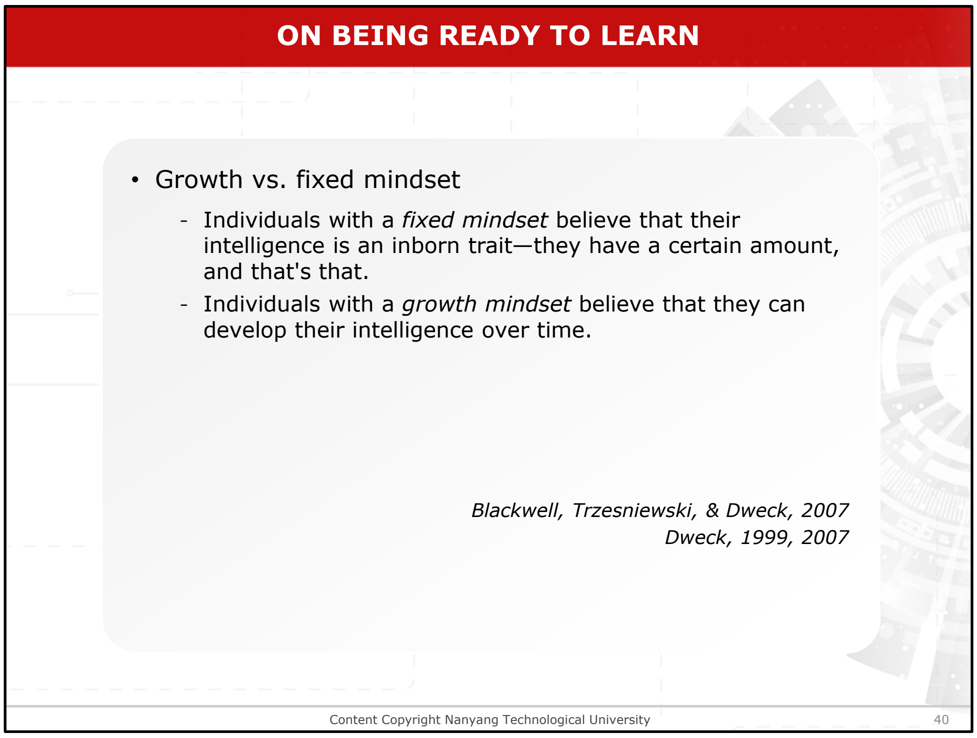


NEXT TIME

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



We will discuss about actual linked lists data structure in the next lecture.

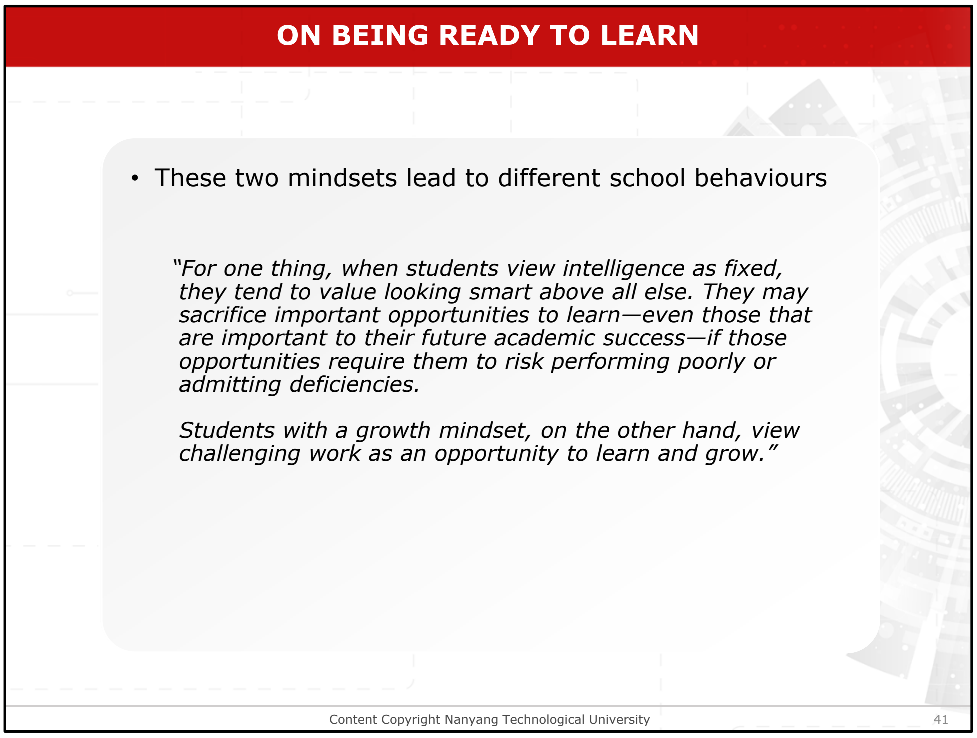


ON BEING READY TO LEARN

- 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

Difference between growth and fixed mindsets.



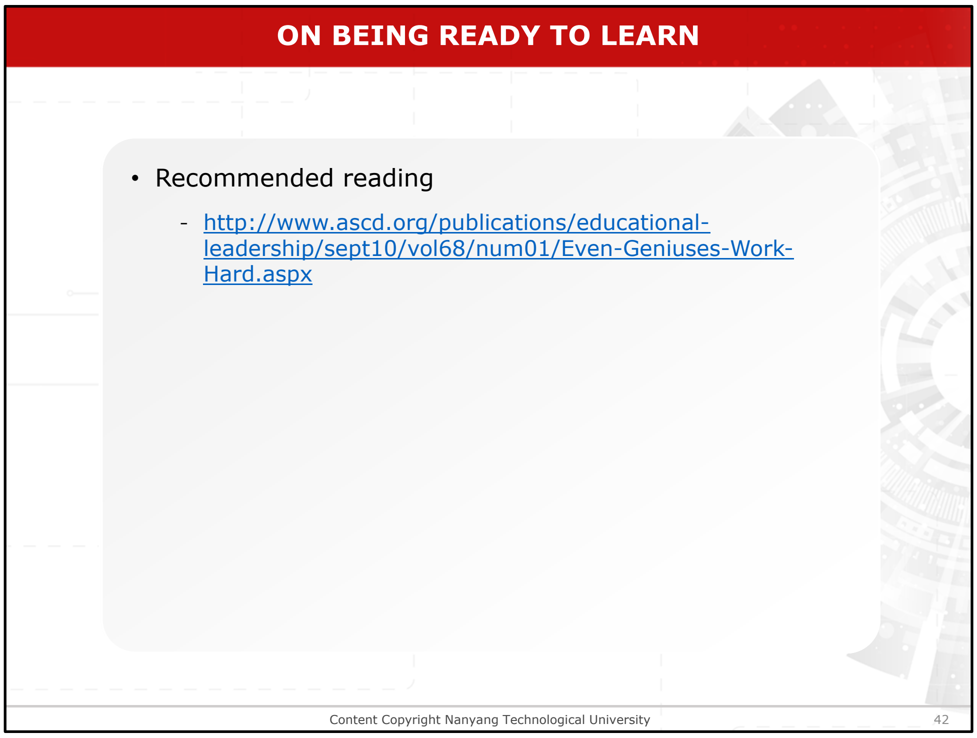
ON BEING READY TO LEARN

- 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."

Growth and fixed mindsets lead to different school behaviors.



ON BEING READY TO LEARN

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

42

Please refer to the recommended reading materials.