

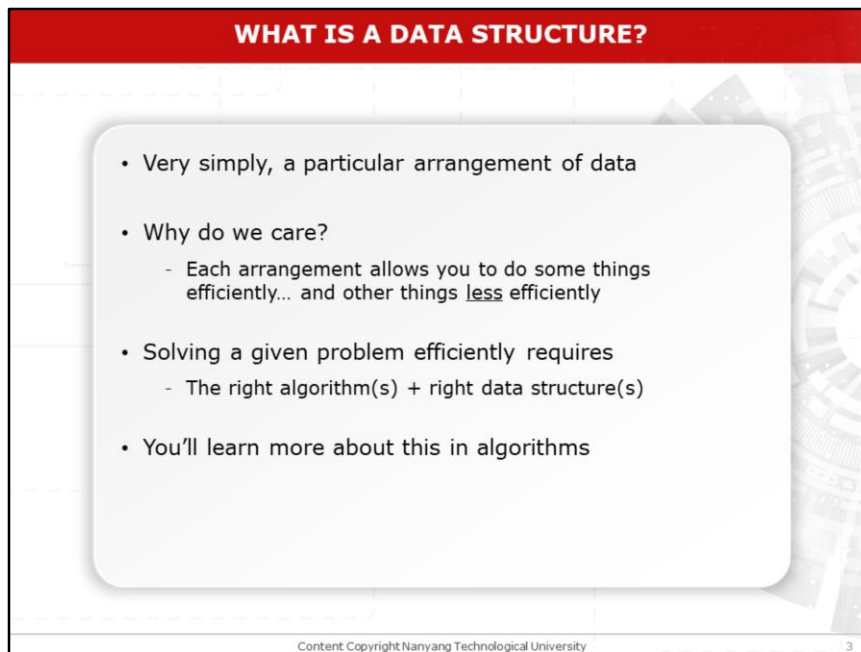
LEARNING OBJECTIVES

After this lesson, you should be able to:

- Explain the difference between static and dynamic elements

Content Copyright Nanyang Technological University 2

After this lesson, you should be able to understand the difference between static and dynamic data structures.



WHAT IS A DATA STRUCTURE?

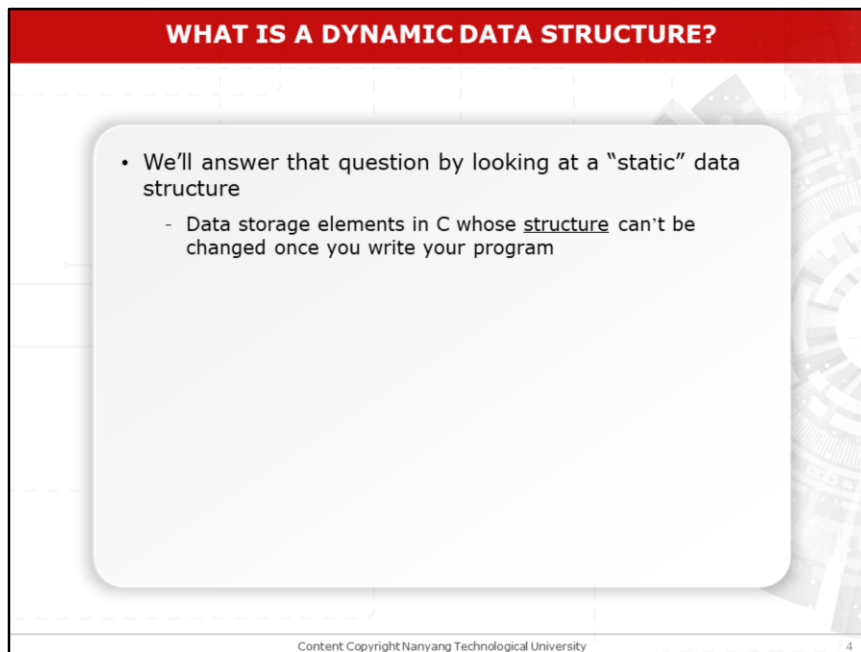
- Very simply, a particular arrangement of data
- Why do we care?
 - Each arrangement allows you to do some things efficiently... and other things less efficiently
- Solving a given problem efficiently requires
 - The right algorithm(s) + right data structure(s)
- You'll learn more about this in algorithms

Content Copyright Nanyang Technological University 3

Data Structure is an arrangement of data in a computer's memory.

A specific way of data arrangement is necessary when building large scale applications on which even a microsecond difference of the run-time matters.

The developer must choose the appropriate data structure and algorithm for better performance.



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

Content Copyright Nanyang Technological University 4

What is a dynamic data structure? Before finding an answer to this question, it is better to understand what a static data structure is.

A static data structure is an organization or collection of data in memory that is fixed in size.

This results in the maximum size needing to be known in advance, as memory cannot be reallocated at run-time.

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};`
- The size of the structure **cannot be changed** once you run the program.

Content Copyright Nanyang Technological University

5

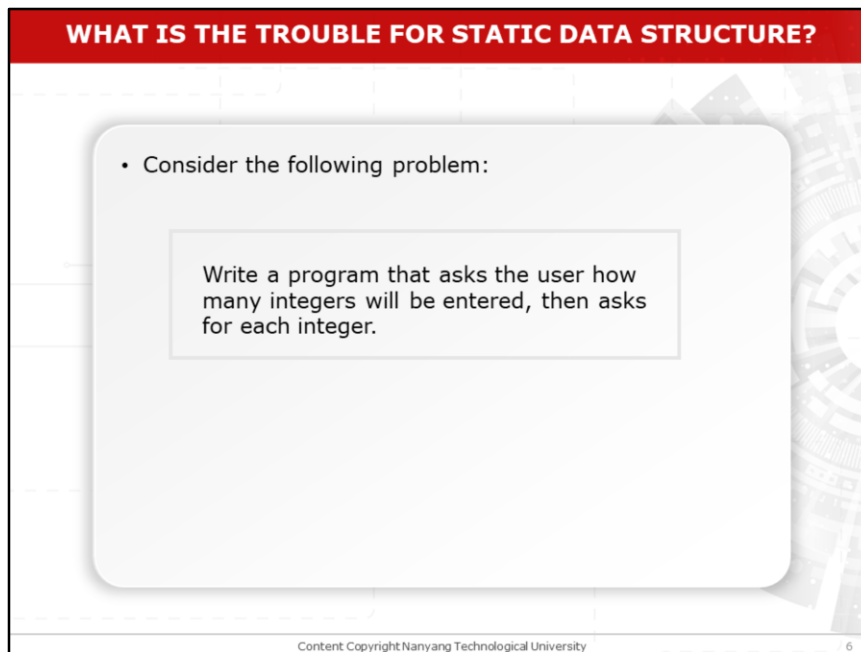
The static data structure is given a fixed area of memory which it can operate within. The content of such data structure can be modified but without changing the memory space allocated to it.

Arrays are a prominent example of a static data structure.

For example, after you define an array of 20 characters, you cannot change the size of the array at run-time.

Even for structs, you have to declare upfront, for instance, struct name and all the member variables inside.

You can't change them once you have compiled the code. You have to make sure that everything is specified before compiling your code.



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.

Content Copyright Nanyang Technological University 6

Static data structures are ideal for storing a fixed number of data items, but they lack the flexibility to consume additional memory if needed or free up memory when possible for improved efficiency.

Let's explain how to overcome such issues through a simple program. Consider the following question;

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

BACK TO THE LABS

- Example code

```

1  #include <stdio.h>
2  #define MY_FAVORITE_BIG_NUMBER 100
3
4  Void main()
5  {
6      int numOfNumbers;
7      int numArray[MY_FAVORITE_BIG_NUMBER];
8
9      scanf("%d", &numOfNumbers);
10
11     for (i=0; i<numOfNumbers; i++){
12         scanf("%d", &numArray[i]);
13     }
14 }

```

Content Copyright Nanyang Technological University

7

Let's look at the sample code written for the given question.

The main problem that you would encounter when trying this question is that you would not be able to predict how many integers the user will enter.

Will it be 100 as suggested in the given code?

How do you handle when the user enter 200 or 500?

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 is smaller than the number of integers that the user is going to enter, the program will overflow the bounds of the array.

The problem is that you need to declare the array size before compilation.

Because your compiler needs to know how much memory space to set aside for every single variable that you have before it can complete compiling your code.

BACK TO THE LABS

- Problems
 - Have to declare array size before compilation

```
#define MY_FAVORITE_BIG_NUMBER 100
int numArray[MY_FAVORITE_BIG_NUMBER];
```
 - 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[232]`
 - Ignore the wasted space

Content Copyright Nanyang Technological University

8

Let's look at the sample code which is focused up to 100 elements.

If you declare an array of 100 integer elements, the memory will set aside space for 100 integer elements which are 400 bytes. After compilation, you cannot change the array size.

Therefore, the solution to this problem is to declare the array with a bigger number. You can declare a maximum size for an integer 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?

Content Copyright Nanyang Technological University

The structures are also very important because structure creates a data type that can be used to group items of possibly different types into a single type.

Let's look at struct declaration more deeply.

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.

I can also declare an array of structs as well. Yet, declaring structs array for 3 elements would limit my program for an array with 3 structs.

That doesn't solve the problem because I'm still limited to a fixed-sized array, I can't grow that array or shrink the array. So we are still back to the same fundamental problem.

What if you want to declare more variables, arrays, or 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 }
```

Content Copyright Nanyang Technological University

10

How to declare an array of integers dynamically?

In the line number five, it says that I need to declare an integer array of size number of numbers.

The idea is that once the user enters a random number, the compiler needs to set an array for the entered number.

The program can then allocate enough space to store the number of integers.

Therefore, you will have an array of integers that is exactly the right size to hold whatever the numbers that the user is going to enter.

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 the integer
9     // Now store the new struct somewhere
10 }
```

Content Copyright Nanyang Technological University

11

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

Let's look at how to allocate memory for structs dynamically.

So I've declared the struct in line number one, that struct called my struct which has one member field. There is an integer called theinteger.

The function has written to store input values.

Every time, a new value comes in, the function requests space for it and stores it inside the requested memory space.

This process will continue until the user enters the sentinel value of -1.

Therefore, I can write the code in such a way that when I need more memory I can request space and when it is given, I can store values.

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 will learn to do in a few more slides
 - You will be doing a lot of this with data structures

Content Copyright Nanyang Technological University 12

Let's discuss about computer memory layouts to understand the static and dynamic memory allocation.

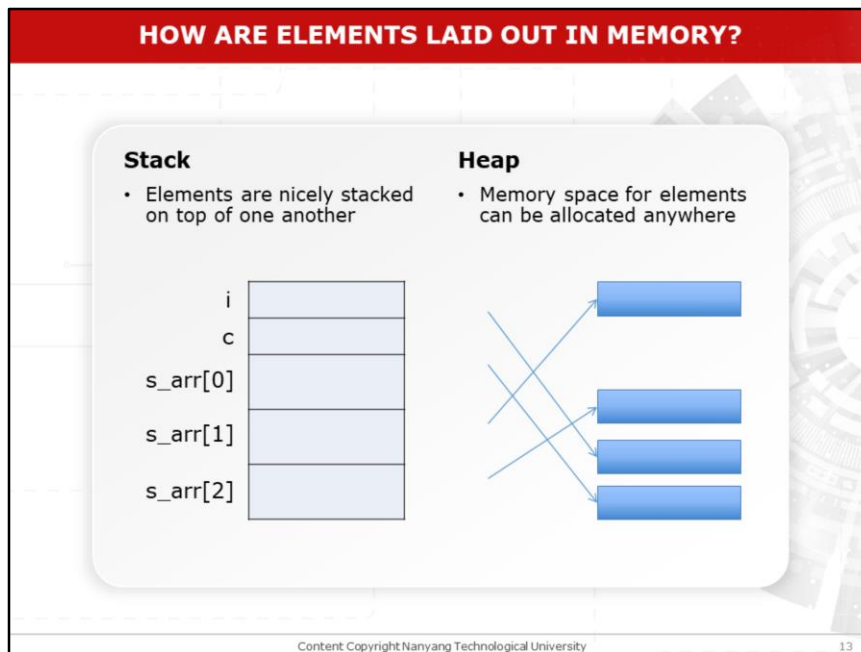
If we have 3 variables as **integer i**, **character c**, and a **struct array of 10 structs**, these variables will be stored in the part of the computer memory called stack.

Stack is used for static memory allocation.

Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and its allocation is dealt with when the program is compiled.

Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower.

Element of the heap have no dependencies with each other and can always be accessed randomly at any time.



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 that the stack will allocate 4bytes, and give access to only 1byte, because the computer memory tries to optimise certain things.

You can also see the first three elements of my struct array that I'd declared. So everything is very nicely stacked and nicely arranged.

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.

You can use the stack if you know exactly how much data you need to allocate before compile time.

You can use heap if you don't know exactly how much data you will need at runtime.

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 for memory allocation on the heap

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

Content Copyright Nanyang Technological University

14

In C, the variable declaration should be done at the top of your code. For an example, you have to declare integer variable at the beginning of your code.

Therefore, to allocate memory in the heap only when needed, C introduced the utility function called '**malloc**'.

'**malloc**' is the short form of memory allocation or memory allocator.

DYNAMIC MEMORY ALLOCATION

- C provides a function for memory allocation on the heap

```
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

Content Copyright Nanyang Technological University

15

When the `malloc()` function is called with the required size, it returns a pointer to a newly allocated block *size* bytes long.

If the requested size of the memory is not available on the heap, the function returns a null pointer.

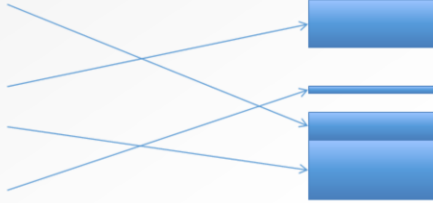
As you can see, the return type for `malloc()` is `void*`.

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typecasted to any type.

For instance, `malloc()` returns `void*` which can be typecasted to any type like `int*`, `char*`, etc.

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



Content Copyright Nanyang Technological University

16

When we request memory from the heap, it grabs whichever free space that is available.

Why would malloc fail?

Let's assume that I have 1GB memory and 750MB is already allocated for different programs, then 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 80MB memory block, then the `malloc()` is going to be failed because the memory chunks are fragmented.

Memory allocation failed for both scenarios hence `malloc()` will return NULL pointer.

Null pointer is a special reserved value of a **pointer**. A **pointer** of any type has such a reserved value. Formally, each specific **pointer** type (`int *` , `char *` etc.) has its own dedicated **null-pointer** value. Conceptually, when a **pointer** has that **null** value it is not pointing anywhere.

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  }

```

Content Copyright Nanyang Technological University

17

How to allocate memory for an integer using `malloc()`?

As you can see, instead of declaring integer `i`, we declare a pointer `i`, to keep track of the allocated memory. 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`).

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, we pass **`sizeof(int)`**, so that the compiler allocates the correct value based on the platform.

Let's look at the coding example;

Line 3 is declaring integer `* i` pointer

Line 4 gets the allocated memory address and assign it to pointer `i`

Line 5 & 6 are for sanity check just in case if `malloc()` fails to perform

Line 7 does the `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 inside the memory.

malloc() BASICS: int

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *i;
7      i = malloc(sizeof(int));
8      if (i == NULL)
9          printf("Uh oh.\n");
10
11     printf("Enter an integer: ");
12     scanf("%d", i);
13     printf("The magic number is %d\n", *i);
14     return 0;
15 }
```

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 }

```

Content Copyright Nanyang Technological University

19

When we allocate memory for an array of 10 integers, we need to pass 10×4 bytes or $10 \times \text{sizeof}(\text{int})$ because now we need space to store 10 integers.

Therefore, as in Line number 7, we pass **$n \times \text{sizeof}(\text{int})$** for n number of integers. And from here, you should be able to write the corresponding code from lines ten onwards, to make use of this array that you declared.

In line number four, you see that I have declared int array as an int star. Here, we declared integer array as a pointer (**int_arr**). We have to do this because the malloc() function returns the memory address which has to store as a pointer.

Yet, in C, arrays and pointers are interchangeable in a certain situation. When you pass an array into a function, you never pass by value; you never pass a copy of the array and its contents. You are always passing arrays into functions by reference.

I have the pointer to the starting address of the integer array, and I have declared **int_arr** as **int_arr** pointer.

Since I have the pointer to the starting address of the integer 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: int ARRAY

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int n;
6     int i;
7     int *int_arr;
8
9     printf("How many integers do you have? ");
10    scanf("%d", &n);
11
12    int_arr = malloc(n * sizeof(int));
13    if (int_arr == NULL) printf("Uh oh.\n");
14
15    for (i=0; i<n; i++){
16        printf("Enter an integer: ");
17        scanf("%d", &int_arr[i]);
18    }
19    return 0;
20 }
```

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 }

```

Content Copyright Nanyang Technological University

21

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), so that it compiles everywhere.

Remember that we use n+1 to leave a space for the slash zero at the end of the array of characters.

malloc() BASICS: string/char ARRAY

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int n;
7      char *str;
8
9      printf("How long is your string? ");
10     scanf("%d", &n);
11
12     str = malloc(n+1);
13     if (str == NULL) printf("Uh oh.\n");
14
15     printf("Enter your string: ");
16     scanf("%s", str);
17     printf("Your string is: %s\n", str);
18     return 0;
19 }
```

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

Content Copyright Nanyang Technological University

And now we can make things a bit more complicated. This is the start of what we are going to be doing for the next five weeks.

The only thing you start off with is that one tiny gray box on the left, which is one pointer variable. Using that one pointer variable, you will keep track of everything else. You can create new blue nodes, new structs, and you can link them to each other or delete blue chunks and so on.

Now, if you look at the given picture, the first blue block is a C struct that we are going to declare later on. As you can see, struct has two parts. The first part can be an integer variable. The second part of that blue chunk is another pointer.

So, I use malloc() to allocate memory for the first struct and the second variable inside the first struct is a pointer which establishes the link to the second dynamically allocated struct. And all of these will be linked up together.

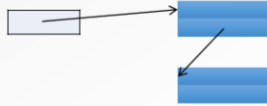
malloc() BASICS: struct TO struct

• **Hands-on:**

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

```

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



The diagram illustrates the concept of a linked list. A small box labeled 'nextstruct' has an arrow pointing to the top of a new blue rectangular block, representing a new struct instance. Another arrow points from the 'nextstruct' member of the first struct to the top of a second blue rectangular block, representing the next struct in the list.

Content Copyright Nanyang Technological University

24

If you look at those two blue chunks over there, each one of those is supposed to be its own struct, an instance of that struct.

The struct is declared up there, struct my struct from line numbers two to five.

It has two members inside, the first member is an integer, so just the number that we are trying to store.

And the second part, which is important because it creates the links between different chunks of data, comes from that pointer.

The type has to be a struct my struct star. Because that link is going to take you to another instance of that struct, so it has to be a pointer to that same type.

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 <stdio.h>
2 #include <stdlib.h>
3 struct mystruct{
4     int number;
5     struct mystruct* nextstruct;
6 };
7
8 int main(){
9     struct mystruct *firststruct;
10
11     firststruct = malloc(sizeof(struct mystruct));
12     firststruct->number = 1;
13     firststruct->nextstruct = malloc(sizeof(struct mystruct));
14
15     firststruct->nextstruct->number = 2;
16     firststruct->nextstruct->nextstruct = NULL;
17     return 0;
18 }

```

The diagram illustrates the memory layout. A box labeled #1000 represents the first struct, which contains a pointer to the second struct at #1010. The second struct is also shown as a blue box labeled #1010.

Content Copyright Nanyang Technological University 25

Let's discuss the coding example in details:

In the code, I have two pointer variables. They correspond to the arrows that I've drawn out on the screen.

The first pointer is **firststruct** which is the pointer to the **struct mystruct**. This pointer has been declared in line 8 and is statically allocated. Therefore, it comes from the stack portion of the memory. Therefore, once you compile your code, it knows that it needs to allocate 4 bytes of memory to hold the address, the pointer variable. That takes me to my very first structure.

After that, from line 10 onwards, everything is being dynamically created. Line number 10 says that firststruct equals to malloc size of struct mystruct, that is the point where it requests enough memory from the heap to store that structure, to store those two member variables inside.

Once malloc() returns with the address in the heap, where you've been given a certain amount of memory to store that structure. And that address is going to be stored into the first struct pointer variable, and that's going to create your very 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. But now look at where this new address gets stored into. It's now being stored into, first struct,

arrow, next struct as you can see in line number twelve.

Let me explain again. I have a pointer variable inside that first struct that I'm accessing. I'm setting the address of my second structure from the heap, at address one zero one zero. I'm storing that address into the next struct pointer, which is inside the first struct.

For instance, once the malloc() returns with the memory address which is #1010 then, we will store that value to the nextstruct pointer variable which is inside the 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.

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 the 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  }
```

Content Copyright Nanyang Technological University 26

When memory is continually being dynamically allocated but not cleared, the computer eventually runs out of memory.

Using malloc() function, you can request a certain number of bytes, and it returns with the address of the allocated memory block.

Using free() function, you can pass the address given by malloc() function and request to clear the memory block which was allocated using malloc() function.

You do not have to pass the number of bytes that need to be cleared because the operating system keeps track of those data.

So now look at this updated function, which is written from line numbers six to eight. In line number seven, malloc() function requests four bytes stores the address where those four bytes start into pointer variable i.

In line number eight, it returns allocated memory back to the system using free() function.

I call malloc() function, so I request four bytes of memory, and then I call free() functions, so I give it back to the system. The total amount of memory that I've held on to is going to remain constant, at any time it's at most four bytes.

EXPLICIT TYPE CASTS

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

This is the standard way to do!

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

- For c, if you include `stdlib.h`, the compiler should automatically take care of everything.
- For cpp, even if you have included `stdlib.h`, you still need to put a type cast in front.

Content Copyright Nanyang Technological University 27

Never forget to include `stdlib.h` when you write standard C code. If you are writing standard C code, use standard C syntax and not C++ syntax.

For C language, if you include `stdlib.h`, the compiler should automatically take care of rest of the stuff.

For C++, even if you have included `stdlib.h`, you still need to put a type cast in front.

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.

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;
5      arr = malloc(5 * sizeof(int));
6
7      for (i=0; i<10; i++)
8          arr[i] = i;
9
10     for (i=0; i<10; i++)
11         printf("%d ", arr[i]);
12 }
```

Content Copyright Nanyang Technological University

28

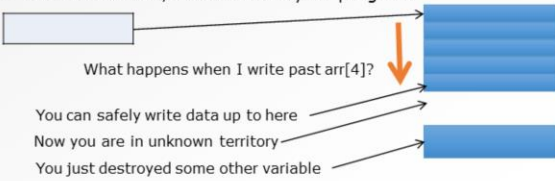
Now, this question is quite interesting because it tests you on your understanding of arrays, how much space you have, and what happens when you start to overflow the bounds of an array.

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 numbers 6 and 7, I am trying to store 10 integers, and in line numbers 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 29

And if you look at the picture over here, we are looking at the heap on the right.

As you 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 zero one two three four, I'm writing into space that actually has been reserved for me. I'm not using somebody else's variable.

But what happens when I start going to array index five, six, seven, eight, and nine?

What am I actually doing? I can still keep going; I can still keep writing to memory. But at some point, I have no idea whether that memory will stay unused. I might be writing into some segment of memory that is already reserved for someone else's variable. I might actually end up destroying other values that were stored there previously. Hence this will end up with unexpected results.

MEMORY LEAKS

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

```

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

30

Content Copyright Nanyang Technological University

When you call malloc() function and forget to call free, that block of memory has already been allocated to you; the system is holding on to it, it assumes that you're making use of it.

Once you finished the function, you will lose the address of the memory block which you previously had.

That memory address is the only way that you can find a way to the block of memory that was given to you. Once you lose that value, that's it; you can't get back to it. This is called memory leak, and it will affect the system very badly.