NANYANG TECHNOLOGICAL UNIVERSITY

CE1007/CZ1007 DATA STRUCTURES

Summary: Dynamic Data Structures

Dr. Owen Noel Newton Fernando

College of Engineering

School of Computer Science and Engineering

MEMORY ALLOCATION IN C

- When you write program you may not know how much space you will need.
- The function `malloc()` is used to allocate a new area of memory. If memory is available, a pointer to the start of an area of memory of the required size is returned otherwise NULL is returned.
- When memory is no longer needed you may free it by calling `free()` function.

Content Copyright Nanyang Technological University

MALLOC()

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

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

- Reserves *size* bytes of memory for your variable/ structure,
  `e.g.,int *i=malloc(sizeof(int)) ;`
- We use the *`sizeof()`* to pass in the correct number of bytes. (ensure correct size on different platforms) ;
- Returns the address (a pointer) where the reserved space starts
  - Returns *NULL* if memory allocation fails

Content Copyright Nanyang Technological University

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   {
4           int *i;
5           i = malloc(sizeof(int));
6           if (i == NULL)
7                   printf("Uh oh.\n");
8           scanf("%d", i);
9           printf("The magic number is %d\n", *i);
10  }
```

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

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

Content Copyright Nanyang Technological University

MALLOC() BASICS: INT ARRAY

- Again, pointer to keep track of array
  - Stores address of start of array
  - Ie, 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

MALLOC() BASICS: STRING/CHAR ARRAY

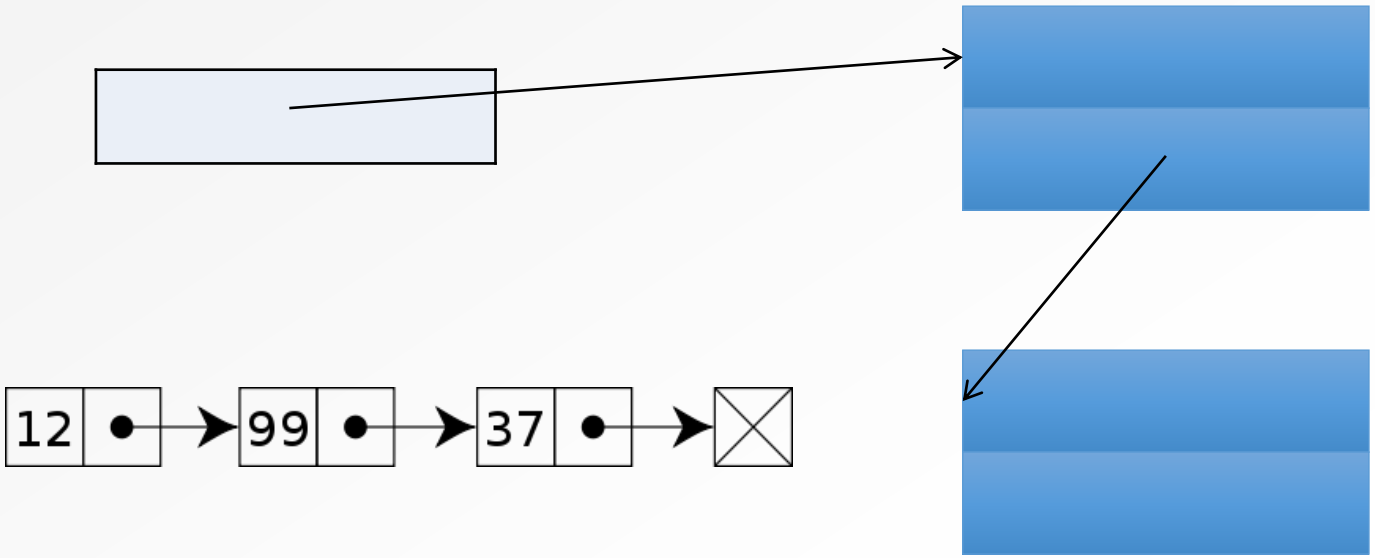
- Same as *int array*, except it's chars
- Allocate n+1 bytes to account for the *\0* string terminator

```
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

MALLOC() BASICS: STRUCT TO STRUCT

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



Content Copyright Nanyang Technological University

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

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

Content Copyright Nanyang Technological University

MEMORY DE-ALLOCATION

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

Content Copyright Nanyang Technological University

FREE() FUNCTION

- 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   void myfunc(){
3           int *i = malloc(sizeof(int));
4           free(i);
5   }
6
7   int main(){
8           myfunc();
9   }
```

Content Copyright Nanyang Technological University