LECTURE 6:

DYNAMIC MEMORY ALLOCATION

COMP1002J: Introduction to Programming 2

Dr. Brett Becker (brett.becker@ucd.ie)

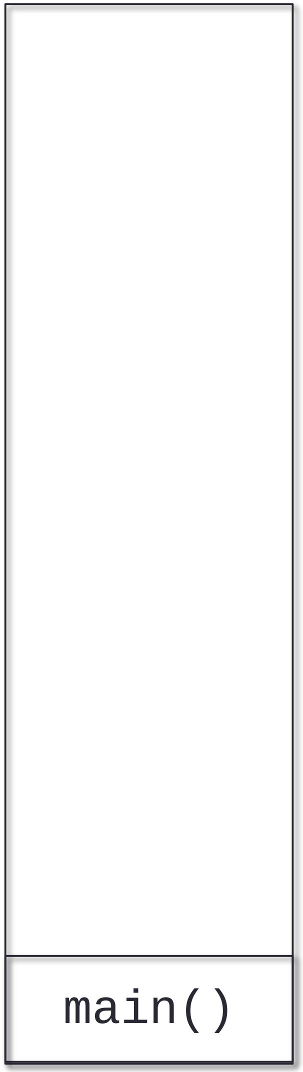
Beijing Dublin International College

Introduction

- Up to now, we have been working with what is known as **statically allocated memory**.
  - All the variables we have used are declared in the program.
  - The compiler allocates (assigns) memory for each variable.
    - The amount of memory allocated depends on the number and types of variables we create.
      - Remember how we calculated the size of a structure.
      - Each time we create a structure, we are allocated that amount of memory.
  - That memory is allocated for the lifetime of the function that it is declared in.
  - While this memory is allocated to the program, it cannot be used by any other programs.

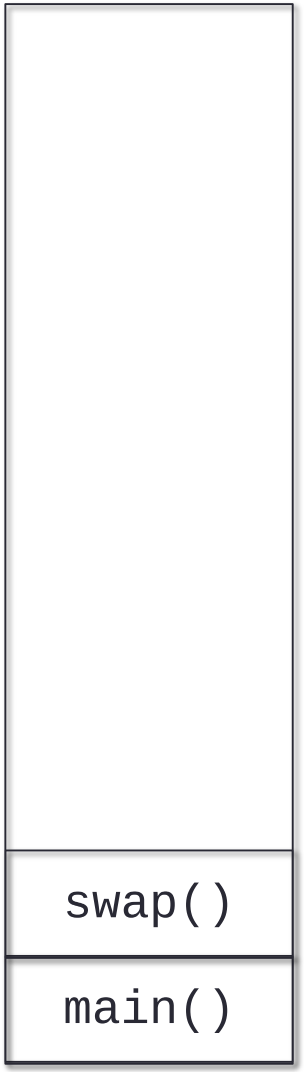
Introduction: Static Memory Allocation

- There are two areas in memory that we can allocate memory in.
- When we use static memory allocation, we are given space in the **stack**.
- Let's imagine that the diagram on the right shows the computer's memory.
- The memory required by `main()` is at the bottom of the stack.



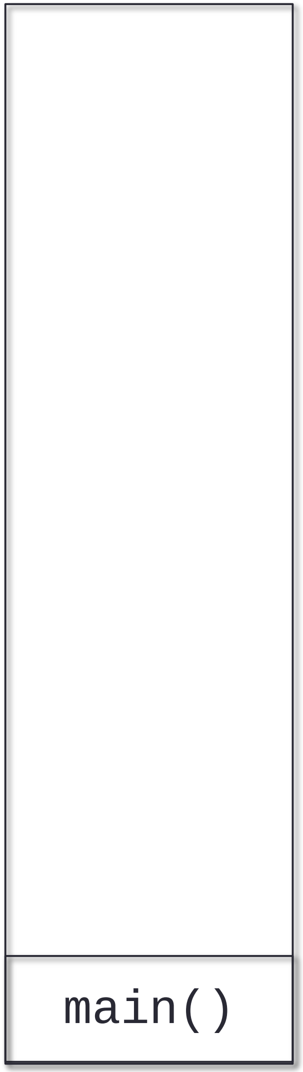
Introduction: Static Memory Allocation

- Now what if we call the function `swap()` within the `main()` function?
- More memory is allocated on the stack, above the memory that belongs to `main()`.
- If more functions are called, then these would also get memory above `swap()`.



Introduction: Static Memory Allocation

- Once the `swap()` function exits, its memory is automatically **deallocated**.
- This means that any variables created within `swap()` disappear and can never be seen by `main()`.
- This happens even if `swap()` returns a pointer to them!



Introduction: Static Memory Allocation

- You should **never** do something like this:

```
Player* create_player(char name[], int age){
   Player p; // <-- this is statically allocated
   strcpy( p.name, name );
   p.age = age;
   return &p;
}
```

- As soon as the function exits, the p variable is **deallocated**.
- It might be given to another function later, in another variable, so the value at that address can change in very strange ways!

Introduction: Static Memory Allocation

- Static memory allocation has a number of drawbacks:

  1. Reusing code is complicated and can often require us to recompile the code (e.g. what if we need an array that has a larger capacity?)

  2. Any memory allocated in `main()` stays allocated until the program ends, even if it has finished using the data (this could be a problem for large programs).

Introduction: Static Memory Allocation

- Static memory allocation has a number of drawbacks:

3. When we write our program, we must say *in advance* **exactly how much memory** we want to be allocated.

4. We must **program defensively**: all arrays should be big enough to make sure that we have enough space to store all the values. This will be a **waste of memory**, as we usually allocate more memory than we need.
   - Remember all of those strings that had lengths like 100, that never actually held 100 characters?

Introduction: Static Memory Allocation

- By the way, this region of memory is called a *stack* because the way that it is accessed is last-in-first-out (LIFO).
  - This means that the last item that is allocated memory is the first item to be deallocated. Which is the same thing as saying that the first item allocated is the last item deallocated.
- The Stack data structure is a LIFO data structure, and that is where the name comes from.
- You will meet the Stack data structure soon.

Introduction: Dynamic Memory Allocation

- C also supports **dynamic memory allocation**:
  - The memory can be allocated (assigned) at run time (while the program is executing).
    - Static allocation is done at compile time.
  - It is done explicitly in the program: you write code that says “set aside X bytes of memory”.
  - When the memory is allocated, the address of the first byte of the allocated memory is returned.
  - This memory address can **only** be stored in a **pointer variable**.
  - *Anything that can be declared statically can also be declared dynamically!*

- Support for dynamic memory allocation is provided through the “`stdlib.h`” library, using the `malloc()` and `free()` functions.

Introduction: Dynamic Memory Allocation

- To allocate memory, we use the `malloc(...)` function.
  - This function takes **1 parameter**: the number of bytes to be allocated (always an integer number of bytes).
  - The **return value** of the function is the address of the first byte in the block of allocated bytes.

- When we allocate memory dynamically, it is allocated on the **heap** (rather than the stack, where static memory is allocated).
  - **Static memory – Stack**
  - **Dynamic memory – Heap**
    - **The heap is also named after a data structure. You'll meet this next year.**

Introduction: Dynamic Memory Allocation

- However, when using `malloc(...)`, the number of bytes we want depends on what we want to store in it.

- On my computer, an `int` is 4 bytes, but I would never write `malloc(4)`.
  - Why not?

Introduction: Dynamic Memory Allocation

- **Answer**: the number of bytes a data type requires can be different for different computers/compilers!

- For this reason, we should **always** use `sizeof(...)` to calculate the number of bytes we want.
  - If I want to get enough memory to store an `int`, I could write:
    `int *i = malloc( sizeof( int ) );`

- We can use this approach with any variable, of any type, even user defined types:

```
float *f = malloc(sizeof(float));
char *c = malloc(sizeof(char));
Player *p = malloc(sizeof(Player));
```

Introduction

```
#include <stdio.h>
#include <stdlib.h>

void staticEg() {
    int i;
    for (i=0; i < 5; i++)
        printf("S%d ", i);
}

void dynamicEg() {
    int *i = malloc(sizeof(int));
    for (*i=0; *i < 5; (*i)++)
        printf("D%d ", *i);

    // we will discuss free() later
    free(i);
}
```

```
int main() {
    staticEg();
    dynamicEg();
}
```

Output:

```
S0 S1 S2 S3 S4 D0 D1 D2 D3 D4
```

file: static_dynamic.c

Introduction: Dynamic Memory Allocation

- So, `malloc(…)` allocates a specified amount of memory and returns the address of the first byte of that memory.
  - This address can be stored in a pointer variable and then used **exactly** like a statically allocated variable (it is accessed via a pointer).

- Let's think about this again:

```
float *f = malloc(sizeof(float));
char *c = malloc(sizeof(char));
```

- Can anybody see what “seems” wrong with this code?

Interlude: Type Casting

- The `malloc(…)` function *seems* to be able to return different types of memory address without problem.

- This goes against how assignment normally works.
  - The value on the right-hand side must be of the same **type** as the variable on the left-hand side…

- What is going on?
  - Basically, C is performing something known as a **type-cast**.

- Type casting is a feature of C that allows you to convert a value from one type to another type.
  - E.g. you can convert a `float` to an `int`.

Interlude: Type Casting

- Example: Convert an `int` to a `float` and vice versa:

```
#include <stdio.h>

int main() {
     float f = 10.0;
     int i = (int) f;
     printf("i=%d\n", i);

     i = 12;
     f = (float) i;
     printf("f=%.2f\n", f);
}
```

Output:

```
i=10
f=12.00
```

file: cast.c

Interlude: Type Casting

- Type casting can be **explicit** (as in the previous example) or **implicit** (the C compiler does it for you).

- In the case of the `malloc(...)` function, **implicit type casting** is taking place.

- The return type of the `malloc(...)` function is actually `void*`
  - `void*` is used to define **a pointer that has unknown type**.
  - This type of pointer is used in cases where you want to be able to store the memory addresses of **different types of data** (we will see this later in the module).

Freeing Memory

- When we use static memory allocation, the memory is automatically deallocated when the function ends.
- **This does not happen for dynamic memory allocation.**
- When we are finished with memory we have dynamically allocated, C provides a way to **deallocate the memory** so that it can be used by other parts of the program.
  - To deallocate previously allocated memory, you use the `free(…)` function, passing a pointer to the memory you wish to deallocate.

```
int *i = malloc(sizeof(int));
// …
free(i);
```

Freeing Memory

- If we dynamically allocate memory, we **must** remember to **always** deallocate the memory when we are finished:
  - `free(pointer_name);`
- If we do not free the memory, then we can create a *memory leak*, where memory is requested but not freed.
  - This can cause the computer to run out of available memory.
- This is not usually a big problem in small programs, but it is a **very important** habit to start doing.
  - If you are programming an embedded system, you might not be able to afford to waste any memory because your resources are limited!
- Programs can crash after a long time because of very small memory leaks that happen often.

Creating Arrays

- We can use the same technique to create arrays dynamically.

- To create an array of integers of size 10, we can use the following code:
  - `int *array = malloc(10*sizeof(int));`

- When the array is created, we can access its values in the normal way (e.g. `array[0]`, `array[1]`, ...)
  - Remember, the name of an ordinary array is just a pointer to the first element of the array. In this example, '`array`' does the same thing.

Dealing with Arrays

- What are the similarities and differences between these two?
  a) `int array1[10];`
  b) `int *array2 = malloc(10*sizeof(int));`

- Both `array1` and `array2` are pointers to the first element of an array.
- They can be used in exactly the same way.
- `array1` is **statically allocated**, so it will automatically be deallocated once the function ends.
  - `array1` can **never** be changed to point anywhere else.
- `array2` is **dynamically allocated**, so the programmer needs to free the memory when finished with it.
  - `array2` is an <u>ordinary pointer</u> and **can** be changed to point somewhere else.

Exercise

- How would I write code to create and dynamically allocate memory for:
  - A pointer to a `long`?
  - A pointer to an array of 10 `ints`?
  - A pointer to a `Player` type?
  - A pointer to an array of 30 `Player` types?

Exercise

- How would I write code to create:

  - A pointer to a `long`?
    - `long *var = malloc(sizeof(long));`
  - A pointer to an array of 10 `ints`?
    - `int *int_arr = malloc(sizeof(int) * 10);`
  - A pointer to a `Player` type?
    - `Player *p = malloc(sizeof(Player));`
  - A pointer to an array of 30 `Player` types?
    - `Player *p_arr = malloc(sizeof(Player) * 30));`

Static and Dynamic Allocation – Differences

- Memory that is allocated statically:
  - Is automatically deallocated (freed) once the function the type is declared in ends
  - Can never point anywhere else

- Memory that is allocated dynamically:
  - Is NOT automatically deallocated
    - Must be deallocated using `free(…)`
  - Can point somewhere else

Disadvantages of Dynamic Allocation

- The downside of dynamic memory allocation is that you must manage the memory yourself.
  - We have already seen that you must manually deallocate memory when you are finished with it.
  - It is also possible that memory allocation will fail due to insufficient space…

- If `malloc(…)` fails, it will return NULL.
  - If the function returns null, then no memory was allocated.
  - You should **always** check the result of the `malloc(…)` function before using it.

Checking for Failure

- Hopefully, `malloc(...)` will not fail often in your programs, but we should still ***always*** check for it.
- Why?
  - If we receive a *null pointer*[1], and try to use it as if it was a pointer to a structure (or something else), the behaviour of the program is then *undefined*.
  - Undefined behaviour means that ***anything*** could happen
    - The program might look like it works perfectly.
    - The program might crash immediately.
    - It might run for a period and crash later.
    - It might appear to run normally, but return incorrect results.
    - It might corrupt data.

[1] https://en.wikipedia.org/wiki/Null_pointer

Checking for Failure

```
#include<stdio.h>
#include <stdlib.h>
int main()
{
        int *ptr_one = malloc(sizeof(int));
        if (ptr_one == NULL)
        {
                printf("ERROR: Out of memory\n");
                return 1;
        }
        else{
                *ptr_one = 25;
                printf("%d\n", *ptr_one);
        }
        free(ptr_one);//why is this here and not inside the else?
```

file: checking.c

Checking for Failure

```
int *ptr_two =
malloc(sizeof(int)*999999999999999999999999999999999999999999999999999999999);
        if (ptr_two == NULL)
        {
                printf("ERROR: Out of memory\n");
                return 1;
        }
        else{
                printf("allocation for ptr_two succeeded!");
        }
        free(ptr_two);
}
```

file: checking.c

Checking for Failure

Output:

```
25
ERROR: Out of memory
```

file: checking.c

Summary

- This means that the recommended basic patterns for using malloc are:

- Allocating memory:
  - Standard Types:

```
<type> *ptr = malloc(sizeof(<type>));
if (ptr == NULL) {
  // indicate failure of malloc
  return(1);
}
//carry on normally
```

  - Array Types:

```
<type> *ptr = malloc(sizeof(<type>)*<capacity>);
if (ptr == NULL) {
  // indicate failure of malloc
  return(1);
}
//carry on  normally
```

- Deallocating memory:
  - All Types: `free(ptr);`

Replace <type> with the type of data (int, float, double, etc.) and replace <capacity> with the number of elements you want in the array.

The realloc() function

- We have just seen how we can allocate memory using `malloc(...)`.
- If we want **more memory** later, we can use the `realloc()` function to reallocate this memory.
- This means our arrays (that are allocated dynamically) can grow!!!
  - This is another difference between arrays that are allocated statically and arrays that are allocated dynamically!

The realloc() function

- `realloc()` will keep the values that were previously in memory.
  - We must pass in a pointer to the old memory that we would like reallocated.
  - And we must also specify the new size of the expanded memory that we want.
    - The new memory can also be smaller if we want.
- `realloc()` returns a pointer to the newly reallocated memory.

```
int *arr = malloc( 20 * sizeof( int ) );
arr = realloc( arr, 40 * sizeof( int ) );
```

The realloc() function

This copies “hello” into str

This concatenates str and “ world!”
Concatenate means ‘joins’

```
char *str;

// Initial memory allocation
str = malloc(6 * sizeof(char)); // don't forget 1 for \n
//check that malloc succeeded here
strcpy(str, "hello");
printf("String = %s,  Address = %p\n", str, &str);

// Reallocating memory
str = realloc(str, 13 * sizeof(char));
//check that malloc succeeded here
strcat(str, " world!");
printf("String = %s,  Address = %p\n", str, &str);

free(str);
```

file: realloc.c

The realloc() function

Output:

```
String = hello,  Address = 0061FF2C
String = hello world!,  Address = 0061FF2C
```