CS 32 Bootcamp

04—Pointers, Arrays, Dynamic Memory

Memory

- Think of it as a contiguous array of bytes
- Each byte has its own **address**
- Every int, array, or object occupies a chunk of memory

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x100																
0x101																
0x102																
0x103																

Addresses/pointers

- Syntax for getting the address of a primitive or object `x`: `&x`
- Syntax for dereferencing the object pointed to by `p`: `*p`
- Arrays are a bit special: they're treated as a pointer to the 0th element!
  - Hence `arr[0]` and `*arr` are the same thing
  - `arr[2]` and `*(arr + 2)` are the same thing
- What does it mean to add a number to a pointer? (**pointer arithmetic**)
  - It depends on the type of the value pointed to
    - e.g. adding 1 to an `int*` causes the address to increase by 4 bytes
- Pointer type: `X*` means a pointer to an object of type `X`

Pointers

- Pointers to pointers: `X**` means a pointer to a pointer of `X`
  - We can have many layers of indirection
- `const`
  - If `const` is before the asterisk, we cannot modify the data being pointed to, but we can reassign the pointer to point to something else
  - If `const` is after the asterisk, we can modify the data being pointed to, but we can't reassign the pointer
  - This is enforced at **compile time**
  - E.g. `const int *p; int *const q; int *const *r;`
  - Question: can we assign a `const int *` to a variable of type `int *`?

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x100																
0x101																
0x102																
0x103																

Arrays

- Remember: an array `arr` is treated like a pointer to the 0th element
- One difference: `sizeof`
  - The size of a static array is the total size in bytes (# elements * bytes per element)
  - The size of a pointer is always 8 bytes (on 64-bit computers)
- When passing an array to a function, the array **decays** to a pointer
  - Thus, inside the function, `sizeof(arr)` will always be 8

Checkpoint: what will this print?

Objects

- Consider this struct:
    - What does `Bar` look like in memory?

```
struct Foo {
    int x;
    int y;
};

struct Bar {
    Foo *fp;
    Foo f;
    char arr[5];
};
```

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x100																
0x101																
0x102																
0x103																

Object pointers

Suppose `p` has type `Bar*` (from the last slide)

- Dereferencing the object: `*p`
- Dereferencing a member of the object: `p->arr`, `p->f.x`, `p->fp->x`
  - Equivalent to `(*p).arr`, `(*p).f.x`, `(*(*p).fp).x`

Objects: Methods

Imagine you are writing C. There is no such syntax as `object.method(arg)`. How would you implement a "method" on an object using only regular functions?

```
struct BankAccount {
    int balance;
};

BankAccount my_account;

// How would you implement:
// The constructor
// my_account.deposit(4)
// my_account.withdraw(9)
```

Objects: Methods

- The answer is: pointers!
- Within a method, there is implicitly a `this` pointer
- Most of the time, you don't need to use `this->member`, you can just use `member`
    - But you must use `this->member` if there is a parameter or local variable named `member`

Function pointers

- Code, like data, lives in memory (von Neumann architecture)
- Like arrays, the name of a function is treated as a pointer
- Calling a function pointer `p`: `p(args)`
- Function pointer type: this is tricky
  - If the function signature looks like this: `ret_t f(arg1_t x, arg2_t y, arg3_t z);`
  - And we want to store its address `&f` in a pointer variable `p`
  - Then we declare `p` like this:

    ```
    ret_t (*p)(arg1_t, arg2_t, arg3_t) = &f;
    ```
- Pointer arithmetic with function pointers is undefined behavior
- Why use function pointers? **Callbacks**

Dynamic allocation

- Sometimes we don't know the array size beforehand, or we might want to avoid allocating a huge object if certain conditions are not met.
- Use the `new` keyword to dynamically allocate an object:

  ```
  X *p = new X;
  X *p = new X(constructor args);
  ```
- Use the `new[]` keyword to dynamically allocate an array:

  ```
  X *arr = new X[array size];
  X *arr = new X[array size]();
  ```

Dynamic allocation

- Dynamically allocated memory must be manually freed!

  ```
  delete p;
  delete[] arr;
  ```

- This will free the memory, but p will still contain an invalid address
  - Dereferencing p after it has been deleted is UB. This is known as a **use after free** bug.
  - Deleting an invalid p is also UB. This is known as a **double free** bug.

Homework

- Homework 1 finalized
- Homework 2 will be out tomorrow, autograder TBD
- Topic: dynamically allocated linked lists