Memory, Pointers, and The Heap

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Contents

- How Does Memory Work?
- Pointers
- Virtual Memory
- Dynamic Allocation

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

How Does Memory Work

- Your programs need to store data somewhere.

- That somewhere is memory!!

- Memory is large store of bytes for you to use.

Memory Addressing

- We store all our data in memory
- We need to be able to access data stored in memory
  - So we can read and write to it

Memory Addressing

- Memory is laid out as a big row.
- You can think of it as an array that takes up every byte of memory.
- An index into that array is a memory address.

Pointers

- A pointer is a variable that stores a memory address
- It is just a number for how far into memory a piece of data is.
- A pointer ‘points’ at the start of the data stored at a memory address.

Pointer Sizes

- Remember that different data types can store different ranges of numbers

- unsigned char – 8 bits
  - 0 – 255
- unsigned short – 16 bits
  - 0 – 65,535
- unsigned long – 32 bits
  - 0 – 4,294,967,295
- unsigned long long – 64 bits
  - 0 – 18,446,744,073,709,551,615

Pointer Sizes

- This means that the size of a pointer defines how high of a memory address you can count up to.
- However, you can't directly set the size of a pointer.
- So how big are they?
- Depends:
  - 32 bit build – pointers are 32 bits
  - 64 bit build – pointers are 64 bits
- The number of memory addresses you could have is called your address space.

The Null Pointer

- The memory address of 0 is reserved to mean an uninitialized pointer - The null pointer.
- In C++ we have a predefined constant to refer to the null pointer called nullptr

Memory address syntax

- In C++ there are three important pieces of syntax for dealing with memory addresses.
- *, & and ->

&

- The & operator is the ‘memory address’ operator.

- Adding it to the front of a variable returns the memory address that the variable is stored at.

```
int my_number;
&my_number; //This is the memory address of my_number
```

*

- The star operator does two things, depending on where its used.
- If used when creating a variable, it modifies the type to be a pointer.
- This works with any type.

```
int my_number;
int* my_pointer; //This is pointer to an int
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

* and &

- The * operator can be combined with the & operator.
- This makes my_pointer point to my_number.
  - In other words, we set my_pointer's value to be the memory address of my_number

```
int my_number;
int* my_pointer = &my_number;
```

*

- We said that the * had two uses.
- The second use is when it is used with an existing pointer variable.
  - It reaches through the pointer and returns the value at the address it is pointing at.
  - This is called 'dereferencing the pointer'
- Dereferencing lets us use the pointer as a regular variable. We can check its value, change its value, pass it to functions, etc.

```
int my_number = 15;
int* my_pointer = &my_number;
std::cout << *my_pointer << std::endl;
```

->

- Lets say we wanted to store a pointer to a struct.
- In order to get access to the elements inside the struct we would have to do this.

```
struct Point{ float x, y; };
Point point = { 5, 3 };
Point* point_pointer = &point;
//dereference, and then access the variable
(*point_pointer).x = 5;
```

- This is such a common task, we have a syntax for access the variables inside a pointer to a struct

- Instead of writing the bottom line, we could have written this

```
//dereference, and then access the variable
point_pointer->x = 5;
```

Canberra Institute of Technology

aie

SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Why do we use pointers?

- Pointers let us do some very, very important things:
  - Pointers let us access the same value from multiple places in our code.
  - Pointers let us store variables that change what data they are referring to.
  - Pointers let us pass memory addresses into functions.
  - Pointers let us access memory that has been allocated dynamically.

Have a break

Pointers

- Accessing the same values from multiple places:
  - Take this Enemy struct as an example.
  - All enemies have a pointer to the player
  - This means they can all easily access the player and make AI decisions based on its state.
    - Can I see the player
    - Does the player have low health
    - How close is the player – should I shoot them or melee
  - Because it is a pointer, we could have 100 enemies, all with access to the same player.

```
struct Enemy
{
    int health;
    float x;
    float y;
    Player *player;
};
```

Pointers

- Have variables that change what data they are referring to:
  - In the example to the right, the player has a pointer to the closest enemy.
    - We could use this to have the player always face the closest enemy
  - Which specific enemy the player is looking at will change as the enemies move around, but this still works as we can change the address the variable is pointing to.

```
struct Player
{
    int health;
    float x;
    float y;
    Enemy *closest_enemy;
};
```

Pointers

- Passing memory addresses into functions:
  - When calling a function, any arguments that get passed are duplicated when they are copied into the parameter variables.
  - This has two main disadvantages
    - If we modify the variables inside the function, it only effects the copy.
    - Its slow to have to copy larger variables.
  - Passing a pointer to the variable solves both problems
    - The pointer still gets copied, but the value stays the same, so we can access the data it points to.

Arrays are Pointers

- Pointers are often used to point to the start of an array.
- When passing an array into a function, it is passed as a pointer to the start of the array.
- The subscript operator '[]' works with a pointer the same way it works with an array.

Pointers

- Accessing dynamically allocated memory:
  - So far all the variables we've used have been stored on the stack, however the stack has some limitations.
    - Is quite small (1MB by default)
    - Does not let you decide how much to allocate at runtime.
  - We can ask for blocks of memory directly from the operating system. This is called dynamic memory allocation.
  - The OS will return us a pointer to the block of memory it allocated.

Dynamic Memory Allocation

- You have a small amount of memory on the stack.
- You aren't allowed to use any of the other memory in the system
  - If you try, your program will crash.
- To get access to any more memory, you need to ask the operating system to give you access.
- This is for a number of reasons
  - All the different programs running need to share the memory on the system.
  - It means that

Dynamic Memory Allocation

- In C/C++ we can ask for dynamic memory in a few ways:
  - Low level OS specific virtual memory system calls (don't use these)
    - Windows – VirtualAlloc
    - OSX, Linux – vm_allocate, mmap
  - C runtime functions (be careful when using these)
    - malloc
    - calloc
    - realloc
  - C++ operators (most of the time, you'll be using these)
    - new
    - new[]

Dynamic Memory Allocation

- Any memory you allocate dynamically ***must*** in turn be de-allocated once the memory is no longer needed.
- Memory is a limited resource – you only have so much of it.
- Allocating memory without de-allocating it when you don't need it any more means you can't use that memory to do other things.
  - If you have code that repeats allocating memory without de-allocating it, that is a memory leak, and can lead to you running out of memory.

Dynamic Memory Allocation

- Just like there are several ways to allocate memory, there are several ways to deallocate it.
  - Low level OS specific virtual memory system calls (don't use these)
    - Windows – VirtualFree
    - OSX, Linux – vm_deallocate, munmap
  - C runtime function (be careful when using this)
    - free
  - C++ operators (most of the time, you'll be using these)
    - delete
    - delete []

Dynamic Memory Allocation

- Every allocation function has a matching de-allocation function

- You should always, ***always*** make sure you only use the matching de-allocation function for the allocation function you called.
  - Do ***not*** mix and match

Summary

- Memory is a resource you have in the system.
- Memory is accessed by addresses
- Pointers let you store a memory address in a variable.
  - Pointers are hugely useful
- You get access to a small amount of system memory on the stack.
- You can get access to more data by allocating on the heap.

References