Memory Management and Pointers in C/C++

By Anirudh Balasubramanian

Create a Memory-Safe Storage Application with C++ on Rhyme

What is Memory Management?
&
Why do we need to know about it?

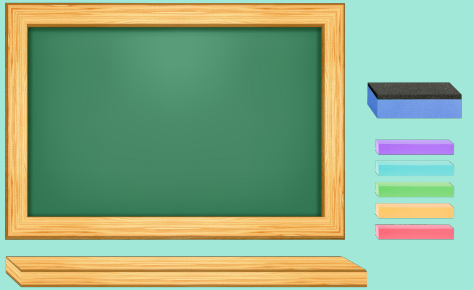
Memory Management Basics

- Memory management is the act of creating and destroying variables in memory (RAM) of a program
- Proper memory management ensures we don't have memory leaks
  - This allows our programs to run for long periods and on a variety of systems
- Some languages have fully automated memory management, each programming language supports a varying level of direct memory access and management
  - C++ is in the middle, allows access but optional automation
- Pointers are the way that memory management is handled in C++
  - As we will see there are multiple different types of pointers
- C and C++ treat pointers in almost identical ways

What is a Pointer?

- This simple question causes quite a bit of concern to many programmers
- A pointer is a variable that stores a memory location
  - In other words, pointers are special integers that hold a memory address
- Memory addresses are stored in HexaDecimal
  - Hexadecimal is base-16, each digit can hold a value from 0-9 or A-F, each place has a value of: (value in space) * 16^(# of spaces from the right)
  - A3 in hex = 163 in decimal
- To get the memory location of a pointer, use a & before the name of the variable
- This location can be used like any other hex number, but we will use it to modify the value of the variable stored at that location

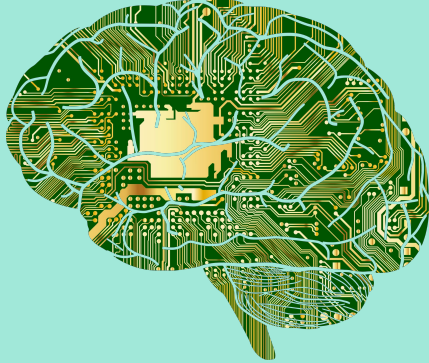
Why do we need Pointers?



- We need them to manipulate arrays
- They can be used to save a great deal of memory and execution time for a project
- Can be used to represent C-Strings
- Pointers to functions
- Essential to implement many data structures
  - Often clearer than the corresponding version in a language that doesn't support pointers
- The stack allows us to use automatically managed memory, but is rather small
  - The heap allows us to dynamically use memory and can be increased to take in large objects/files we need for our programs
- We will use them to build a Linked List but pointers are useful in many C++ and C applications

Basics of Using Dynamic Memory in C++

Using Memory Management in C/C++



- For now, we will discuss normal C/C++ pointers
- To get the address of any type of variable, place a “&” before the name of the variable
  - For example to get the address of int x: &x
- To create a pointer place a “*” before the variable name when it is created
  - For example: int *xPtr = &x;
  - To declare multiple pointers on the same line, place a * before each variable’s name (not just one per line)
    - Int *xFirstPtr, *xSndPtr, *xThrdPtr;

Using Memory Management in C/C++ Cont.

- New
  - New is how we allocate new dynamic memory in C/C++
  - It indicates that we want to create space in memory for a new variable
    - This memory stays allocated until we call delete
      - If delete is never called, the memory has leaked, and isn't usable by other programs until our program terminates
- Delete
  - This allows us to free up memory as we know a variable is no longer in use
  - Simply say delete and then the name of your variable
    - Delete intPtr;
  - This memory is now junk data, so if we try and access it we will have a segmentation fault

Pointer Manipulation

Pointer Manipulation

- Null
  - After deleting dynamic memory, the location becomes junk data
  - To properly clean up the pointer, set it equal to null or nullptr or 0 (all equal)
  - We can then check if a pointer is null as a condition to see if we reached the end of a data structure
- Arrays
  - Arrays and pointers are closely tied in C/C++
  - All arrays in C/C++ are special groups of pointers
  - When we delete a array we must use the delete[], however the name of the array can be used like a pointer in most other places
- Pointer Arithmetic
  - Pointer arithmetic is useful for arrays, we can add a certain number to the pointer and get to a index in the array “X” spaces further than the original first spot
- C-Strings
  - C-Strings are how C stores strings, they are special arrays of chars that have an automatically added null char at the end of the array
  - We also can get access to CStrings in C++ using the CString library
  - C-String to normal string conversions are supported in C++

The Big 3

The Big 3

- The Big 3 is a rule in C++ that applies to standard pointers, that if a class defines any of the following it should explicitly define all 3
  - The destructor (what is called when delete is executed)
  - Copy constructor (How to copy by value for the variable)
  - Copy Assignment Operator (For assigning one object to another)
- These are called special member functions
  - If we use any of them without explicitly defining it, there is always a default method of implementation based on your compiler
  - Destructor
    - Will call all object's class type variable constructors
  - Copy Constructor
    - Construct all the object's members from the corresponding members of the copy constructor argument, calling the copy constructors of the object's class-type members, and doing a plain assignment of all non-class type
  - Copy Assignment Operator
    - Assign all the object's members from the corresponding members of the assignment operator's argument, calling the copy assignment operators of the object's class-type members, and doing a plain assignment of all non-class type data members.

The Big 5

The Big 5

- After C++11, two more methods were added onto the Big 3 to ensure proper functionality, these last two aren’t as regularly needed as the first 3, but are still used quite often
- If we define any of the big 3, we can't implicitly define the move constructor and the move assignment operator,
  - Thus, any class we want move semantics for must declare all the additional two methods in the Big 5

```
39
40      // assignment operator for move ctor
41      EntiTypeA& operator= (EntiTypeA&& rvEntypeA)
42      {
43          if (this == &rvEntypeA) return *this;
44          *this = std::move(rvEntypeA);
45          return *this;
46      }
47
48
49      // move constructor
50      EntiTypeA(EntiTypeA&& mvTypeA)
51      {
52          std::cout << " EntiTypeA move constructor called ! " << std::endl;
53          if (this == &mvTypeA) return;
54          *this = std::move(mvTypeA); // calls to the rvalue move-assignment operator
55
56      }
57
```

An Example with the Big 5

Smart Pointers

Smart Pointers

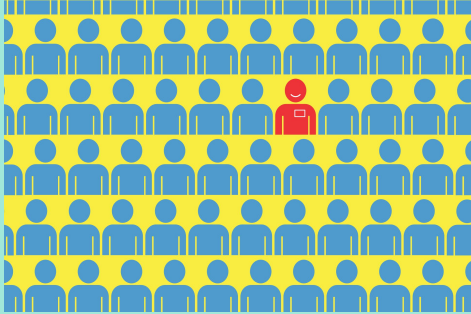
- Smart pointers are a wrapper class around pointers
- They allow for some automatic memory management
  - Today, we normally use smart pointers whenever possible
- With smart pointers, we reduce the need for the Big 3/5 as much more memory management is now automatic
  - We might still need to override the copy assignment and copy constructor but this depends on the exact case we are working with

Types of Smart Pointers

- 3 main types of smart pointers
  - Unique pointers
    - Like normal pointers but can only be referenced by one pointer, when this goes out of scope, the pointer is destroyed and the memory freed up
    - Needs extra memory compared to standard pointers since it is managing the wrapper class and the normal pointer
  - Shared pointers
    - Keeps a counter of how many times the pointer is referenced, when this counter hits 0, the pointer is destroyed
    - Adds extra memory overhead compared to normal pointers due to tracking number of references
  - Weak Pointers
    - A type of shared pointer where even if the weak pointer is still in scope, if all other references are destroyed, then the pointer is destroyed

Special Pointers and Debugging with Pointers

Special Pointers



- Void Pointer
  - Void pointer is a special pointer that you probably won't run into that often
  - Void *ptr = &someInt;
  - A generic pointer that can be pointed at any data type, can be nullptr
    - While this seems ideal, we can't actually use the pointer (dereference it) until we cast the pointer to the right pointer type
    - As a result, we often have to keep track of what type of pointer it should be manually, and it's usually easier to just let C++ handle this
- Pointers to Functions
  - Just like we can have a pointer to a variable we can have a pointer to a function
  - We just create the void pointer and then give it a value of the address of some function in the program
  - The pointer can then be used to call the referenced method (doesn't need to even be dereferenced)

Differences between C and C++ Pointers

- C and C++ pointers are largely the same but there are some small differences
  - C doesn’t require explicit casting for pointer while C++ does
  - C++ allows for pointer to be used automatically such to reference a virtual function
  - C also doesn’t have support for smart pointers so we must manually manage memory or create our own wrapper class
- Otherwise, C and C++ pointers are the same and both treat memory allocation specially

Debugging with Pointers

- Debugging with pointers and dynamic memory can be one of the most frustrating experiences as a programmer
- Using tools like VS Code's debugger and trying to plan the logic of our code clearly before coding are great ways to reduce the errors produced by our programs
- We can place a breakpoint before the problematic code, often when we dereference a pointer
- Stepping through a program and tracing its logic can be powerful too
- Print statements are still valuable
- Make sure you set any dangling pointers to nullptr whenever possible

Hope you enjoyed the course!

Please leave feedback so I can continue to improve my courses

- Future Goals after this Course
  - Try my other courses, if you're looking to solidify your understanding of C++
  - Try and build another data structure with pointers like a Binary Search Tree
  - Build a program with and without smart pointers to see the difference